BETTY WG3 - Languages
State of the Art Report

Giuseppe Castagna Luca Padovani Nobuko Yoshida (editors)

October 25, 2013

Contents

List of Contributors

- Davide Ancona
- Viviana Bono
- Mario Bravetti
- Joana Campos
- Nils Gesbert
- Elena Giachino
- Raymond Hu
- Einar Broch Johnsen
- Francisco Martins
- Viviana Mascardi
- Fabrizio Montesi
- Nicholas Ng
- Rumyana Neykova
- Luca Padovani
- Vasco Vasconcelos

1 Introduction

The study of behavioural type theories as pursued by *WG1 - Foundations* is a mandatory step towards the adoption of behavioural types into mainstream programming languages and the design of new languages features based on behavioral types. Nonetheless, such studies are often carried out on abstract and simplified computational models which do not always capture all practical and idiomatic facets of real-world programming languages. The purpose of *WG3 - Languages* is to fill the gap between theory and practice of behavioral types and to develop principles and key runtime mechanisms for programming with behavioural type systems as intrinsic structures.

Despite the recent interest in behavioral types, there is already a rather massive bibliography on their integration in concrete programming languages and paradigms. This report aims at providing a first comprehensive view on the efforts done so far in this respect and to aid interested researchers at identifying the areas as yet unexplored.

The report is structured as follows:

- Section 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 behavioural 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 interpreting objects themeselves as the entities for which a behavioral description is required, for example to specify the order in which methods must/can be invoked.

- Section 3 explores the integration of behavioral types within other programming contexts, in particular the functional and multiagent paradigms. 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. Behavioral types also provide an effective abstraction tool for describing autonomous entities in multiagent systems. High-performance computing often relies on parallel processes that synchronize by means of message passing. Also in this case, behavioral types provide an effective means for making sure that such communications occur without errors.

- Section 4 provides an overview of the use of behavioral types in Singularity OS, a prototype Operating System developed by Microsoft which 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* closely related to the concept of session type.

- Section 5 describes the potential of behavioral abstractions akin to behavioral types in the specific domain of Web Service description. These descriptions enable sophisticated forms of static and dynamic verification that in turn can be used for implementing runtime discovery and composition/orchestration of Web Services.

- Section 6 explores the potentials of the *design-by-contract* methodology to 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 can then be projected for describing the local behavior of the network participants.

This brief overview already hints at the abundance and diversity of programming contexts in which behavioral types can play a role. In an attempt to make it easier for the reader to appreciate similarities and differences between contexts, all the sections that follow refer to the same simple scenario depicted in Figure 1 and taken from [CDLPRIMER]. In this example there

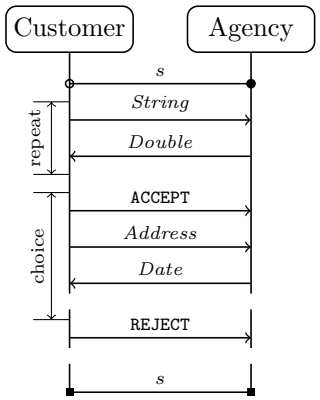


Figure 1: Graphical representation of the Customer-Agency protocol.

are two interacting entities, named Customer and Agency, that establish a communication session s. The interaction consists of two phases: the first one, marked as "repeat" in the figure, is made of an unbound number of queries issued by 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, 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, while the two s-labeled arcs respectively denote initiation and closing of the session between Customer and Agency. The discontinuity in the vertical development of the protocol is meant to suggest that the subprotocols beginning with the `ACCEPT` and `REJECT` messages are mutually exclusive, the decision being taken by Customer.

The sections that follow include a general introduction to the interest and main difficulties of integrating behavioral types to the language/paradigm being considered, the implementation of relevant parts of the Customer-Agency example, either in the form of actual code or as terms of a suitable formal calculus, the peculiar characteristics of the language/paradigm, as well as a short commented bibliography on the approach with hints to open issues and future work.

2 Object-Oriented Languages

2.1 Calculi

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 his 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 encountered with the previous approaches, taking into account the aim of writing concurrent and distributed applications with a better structure and, consequently, more solid.

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], SAM^{g} Capecchi et al. [2009], and SAM$^{\vee}$ Bettini et al. [2008a, 2013].

2.1.1 MOOSE dialects

The work MOOSE (Multi-threaded Object-Oriented calculus with Sessions) is the result of the embedding of 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 to consistently integrate 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.

We will present the language MOOSE by 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. The class `Customer` contains a method `buy` for the purchase of a travel tricket according to the parameters `journeyPreferences` and `maxPrice`. When the method is called, the `connect` on the channel `c1` with session type `placeOrder` is executed. Therefore, 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 placeOrder = begin.!<!String.?double>*.!<!Address.?Date.end,end>
```

The code of a compatible agency is as in Figure 3. 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

```
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); // implem. 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: The class `Customer`.

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

The code of the communication can be as follows:

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

Let us see how this expression reduces following the semantic rules of MOOSE, which can be found in Dezani-Ciancaglini et al. [2009]. During the evaluation, we need additional information about objects and channels, that are recorded in the heap h, defined as follows:

$$h \quad ::= \quad [\,] \;\mid\; h :: [\mathsf{o} \mapsto (\mathtt{C}, \overline{\mathsf{f} : \mathsf{v}})] \;\mid\; h :: \mathsf{c}$$

where :: denotes heap concatenation.

So, the initial configuration is

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

At first we evaluate the `spawn` expression that makes its body become a new parallel thread:

```
new Agency.sell()‖ new Customer.buy("London to Paris,
Eurostar",300)
```
[]

At this point the next expressions to be evaluated are the two `new`. They will create two objects, instances of classes `Agency` and `Customer` respectively, and the heap will be updated with the new information: for each object the heap contains the name of the instance associated to the corresponding class and the names and initial values of the fields:

```
o1.sell()‖o2.buy("London to Paris, Eurostar",300)
```

[o1 ↦ (Agency,journeyPref:"")] :: [o2 ↦ (Customer,addr:"",price:"",loop:true)]

At this point the two methods can be invoked. Notice that the formal parameters in the body of method `buy` are replaced with the actual parameters `"London to Paris, Eurostar"` and 300, and `this` is replaced with the corresponding object identifier:

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

[o1 ↦ (Agency,journeyPref:"")] :: [o2 ↦ (Customer,addr:"",price:"",loop:true)]

```
class Agency {
  String journeyPref;
  void sell() {
    connect c1 acceptOrder {
      c1.receiveWhile {
        journeyPref := c1.receive;
        double price := getPrice( journeyPref ); // implem. 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 3: 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 4: The class `Service`.

Now, since the two `connect`s are on the same channel `c1` with dual session types, the connection can be established and the previous parallel composition reduces to:

```
...                                                  ||  ...
k1.send( "London to Paris, Eurostar" );              ||  journeyPref := k1.receive;
price := k1.receive;                                 ||  k1.send( price );
```

`[ o1 ↦ (Agency,journeyPref:"") ] :: [ o2 ↦ (Customer,addr:"",price:"",loop:true) ] :: k`

where `k1` is a new fresh channel replacing the old channel `c1`, that has been created and added to the heap. The freshness of `k1` guarantees privacy of the session communication between the two threads.

At this point the exchange of data can begin and the parallel composition above reduces to:

```
...                                  ||  ...
price := k1.receive;                 ||  journeyPref := "London to Paris, Eurostar"
                                     ||  k1.send( price );
```

`[ o1 ↦ (Agency,journeyPref:"") ] :: [ o2 ↦ (Customer,addr:"",price:"",loop:true) ] :: k`

Choices. Choices in MOOSE are modeled with the following constructs

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

Similar constructs are used to model iterations: `c.sendWhile(e){e1}` and `c.receiveWhile{e2}`, where the evaluation of `e1` and `e2` is repeated while the result of `e` is `true`.

In Figures 2 and 3 we can see in use both these constructs. Notice that in the original protocol the choice is modeled with a label-based construct, so that the two braches are labeled with specific ACCEPT and REJECT labels, and the `Customer` chooses one branch over the other by sending the corresponding label to the `Agency`. While in here the `Customer` sends just a Boolean value. For more comparison, see the **Peculiarities** section below.

Delegation. Delegation works as in the 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 on `c` and bound by `x` within the expression `e`.

In figure 3 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 calculus in a way to be as close as possibile to the original π-calculus based sessions. Therefore most of the features are mere adaptions of the corresponding ones in the π-calculus. A few differences, however, can be noticed. First of all, choices are based on boolean values instead on labels, in order to be more close to the standard programming constructs and habits. Notice that this feature does not affect expressivity in any way, since it corresponds to a binary labeled choice, thus choices with more branches can be easily encoded with nested expressions, but with the disadvantage of sending multiple boolean values instead of one label.

Other differences are rather technical and concern the use of channels for the connection and communication.

In the original π-calculus sessions Yoshida and Vasconcelos [2007] two parties connect over a public channel, thus the connection contructs mention the public name and a varible binding the new private name in the scope of the session. In here there is no such variable, intead the new private name will replace the public name (e.g. `c1` in Figure 2) in the body of the connection.

Delegation, i.e., the moving of channel names around, does not get along easily with the structural essence of session types. If wrong configuration are allowed then subject reduction may fail Yoshida and Vasconcelos [2007]. Two main solutions have been adopted: one can is based on the use of two different private names, identifying the two endpoint of the communication, another one is based on the use of just one private name with the restriction α-conversion is implicitly performed ahead of communication. This calculus 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 wrong configurations cannot occur.

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 the 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. Let us consider the following session descriptor:

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

that 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. 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 fact suggests that both could be derived from a more general notion of session associated to an object. In Drossopoulou et al. [2007] a language that *amalgamates* the notion of session-based communication with the one of object-oriented programming is proposed. The approach was called STOOP

	"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	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

Figure 5: "Traditional" sessions, "traditional" methods, and "amalgamated" sessions/methods.

(Session Types and Object-Oriented Programming). Stoop is only a "language kernel", since 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.*.

Stoop drives the amalgamation very far, by unifying sessions and methods, and by basing the choices of alternative paths in communications on the object being sent or received (see 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 calculus 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 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, the 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 Figure 5, 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 the Ada rendez-vous, 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

```
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); // implem. omitted
    };
    sendCase( evalPrice(price,maxPrice) ) {
      ACCEPT ▷ send( addr ); Date date := receive;
      REJECT ▷ null; /* customer rejects price, end of protocol */ }
  } /* End method invocation */
```

Figure 6: The class `Customer`.

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

Figure 7: The class `Agency`.

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.

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 6. 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 see the evaluation of the expression above, we need to define the *heap* that stores objects and values exchanged during the communication:

$$h \ ::= \ [\,] \ \mid \ h :: \mathsf{o} \mapsto (\mathtt{C}, \overline{\mathsf{f} : \mathsf{v}}) \ \mid \ h :: \mathsf{k} \mapsto \tilde{\mathsf{v}}$$

where $::$ denotes heap concatenation, k are the private channels created at runtime.

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

Figure 8: The class `Service`.

Now we can see how the invocation is evaluated. At first the heap is empty:

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

A new object of class `Agency` is created and the heap is updated with the information related to the new object:

```
o.sell {
  ...
  send( "London to Paris, Eurostar" );
  price := receive;
  ...
};
```
, [o ↦ (Agency,journeyPref:"")]

At this point the session is invoked and a new thread is spawned in order to execute the body of the session. A pair of fresh channels `k` and $\tilde{k}$ is created, that 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. The heap is updated with an empty queue for each channel, in which the received value will be put, waiting to be read.

```
...                                          ...
k.send( "London to Paris, Eurostar" );   ||  journeyPref := k̃.receive;
price := k.receive;                          ...
...                                          k̃.send( price );
                                             ...
```

[o ↦ (Agency,journeyPref:"")] :: [k ↦ ()] :: [$\tilde{k}$ ↦ ()]

The communication begin with the sending of the first value. The value is put in the heap, 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.

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

[o ↦ (Agency,journeyPref:"")] :: [k ↦ "London to Paris, Eurostar"] :: [$\tilde{k}$ ↦ ()]

Now the result can be read from the queue and stored into the field `journeyPref`. The queues associated to the channels ensures 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:

- `sendCase`(e){$\mathtt{C}_i \triangleright \mathtt{e}_i\}_{i\in I}$: where first the expression e is evaluated and reduced to an object, then depending on its class, if it $\mathtt{C}_i$, it continues with $\mathtt{e}_i$;
- `receiveCase`(x){$\mathtt{C}_i \triangleright \mathtt{e}_i\}_{i\in I}$: receives an object and continues with $\mathtt{e}_i$ if the class of the object is $\mathtt{C}_i$.

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

Similar constructs are used to model iterations: `sendWhile`(e){$\mathtt{C}_i \triangleright \mathtt{e}_i\}_{i\in I}$ and `receiveWhile`(x){$\mathtt{C}_i \triangleright \mathtt{e}_i\}_{i\in I}$, where a special variable `cont` occurring in some of the $\mathtt{e}_i$, when encountered, make the execution start again.

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:

$$\mathtt{e}\bullet\mathtt{s}\{\ \},$$

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 7 where the session is being delegated to a new `Service`: In the class in Figure 7 we see a delegation request to the session `offerDelivery` of `Service`.

That code gets executed if the `Customer` accepts the purchase (buy 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;
```

$$[\ \mathtt{o} \mapsto (\mathtt{Agency},\text{-})\ ] :: [\ \mathtt{k} \mapsto ()\ ] :: [\ \tilde{\mathtt{k}} \mapsto ()\ ]$$

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 stored in the heap, 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 $\tilde{\mathtt{k}}$.

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

$$[\ \mathtt{o} \mapsto (\mathtt{Agency},\text{-})\ ] :: [\ \mathtt{o1} \mapsto (\mathtt{Service},\text{-})\ ] :: [\ \mathtt{k} \mapsto \mathtt{addr}] :: [\ \tilde{\mathtt{k}} \mapsto ()\ ]$$

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 communication is left and the session ends.

Union types. Choices based on the exchanged object, 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 case 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 calculus 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 descriptor

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

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 $\mathsf{SAM}^{\mathsf{g}}$, a core object-oriented calculus 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 means 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 descriptor !`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 Java

In this section we review attempts to integrate a form of behavioural 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 behaviour conformance. The features shared by all of them are: only objects of some specific classes are controlled by the behavioural type system; aliasing is disallowed for these objects; behavioural 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 `http://code.google.com/p/sessionj/` is the largest software 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 (see the companion WG1 report on Foundations of Behavioural Types for a history and references).

SessionJ Hu et al. [2008, 2010], Ng et al. [2011], Alves et al. [2010] is based on the MOOSE calculus described in Section 2.1, with adaptations to Java and additional features; in this language, session-typed communication 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 [2008], Militão and Caires [2009] allows adding a usage protocol to any class, but has no specific construct for channels or concurrency.

Bica Gay et al. [2010], Caldeira and Vasconcelos [2011] 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] has usage protocols and concurrency, but no explicit channels: message-passing between threads is done through regular method calls.

We now review SessionJ and Bica in more detail. Yak's type system is very similar to Bica'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 Bica, with the addition of qualifiers to control aliasing.

2.2.1 Session Java

SessionJ (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: Hu et al. [2008], Hu et al. [2010], Ng et al. [2011], Alves et al. [2010].

It is based on the Moose calculus described in Section 2.1.1, thus the implementation of the running example will be very close; we follow the same structure. In SJ, the session type for the customer side of the protocol is declared as follows:

```
protocol placeOrder {
  begin.
  ![
     !<String>.
     ?(Double)
  ]*.
  !{
     ACCEPT: !<Address>.?(Date),
     REJECT:
  }
}
```

Contrary to Moose, SJ allows labelled branching and not just boolean branching. The other differences are cosmetic. $![S]*$ represents a loop where, at each iteration, this side decides whether to loop or to continue. $!\{L_1 : S_1, \ldots L_n : S_n\}$ represents a choice where this side decides which option to select.

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. In our example, class Customer implements a client. The `buy` method of this class can be written this way:

```
public void buy (String journeyPref, double maxPrice) {
  boolean decided = false;
  SJServerAddress agency = SJServerAddress.create(placeOrder, host, port);
  SJSocket c = SJSocketImpl.create(agency);
  c.request();
  c.outwhile(!decided) {
    c.send(journeyDetails);
    double cost = c.receive();
    decided = evalOffer(journeyPref, price);
  }
  if (price <= maxPrice) {
    c.outbranch(ACCEPT) {
      c.send(address);
      Date dispatchDate = c.receive();
    }
  } else {
    c.outbranch(REJECT) {}
  }
```

where `placeOrder` has been declared previously as described above. Here `c` is seen as 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 `c` must be created inside the `buy` method and cannot escape it (except by being *delegated*, which will be illustrated later). In particular, it cannot be a field of the Customer class.

Apart from the part establishing the communication, the code is reasonably close to Moose. `send` and `receive` look like regular Java method calls. The syntactic construct `outwhile(bool) {loop body}` corresponds directly to the `sendWhile` of Moose; the analogue of Moose's `sendIf` on the other hand is a combination of a regular java `if` and two `outbranch(label) {body}` calls to select a particular label and then execute the body part. Since the branching is not done by a specific construct here, a n-ary choice could be implemented as a cascade of if/elses, as a switch/case, or as 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`.

The dual protocol which the agency must implement is `acceptOrder`, declared as follows.

```
protocol acceptOrder {
  begin.
  ?[
     ?(String).
     !<Double>
  ]*.
  ?{
     ACCEPT: ?(Address).!<Date>,
     REJECT:
  }
}
```

Delegation As in the Moose example, we illustrate delegation by having class Agency delegate the end of the transaction to a remote service after the customer has selected ACCEPT. This delegation takes place itself over a session-typed channel; the type of this channel is declared thus:

```
protocol delegateOrderSession {
  begin.!<?(Address).!<Date>>
}
```

on the Agency side and thus:

```
protocol receiveOrderSession {
  begin.?(?(Address).!<Date>)
}
```

on the Service side.

The `sell()` method of class Agency is then:

```
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).

The delivery() method of class Service is::

```
void delivery() {
  SJServerSocket ss_sa = SJServerSocketImpl.create(receiveOrderSession, port);
  SJSocket s_sa = ss_sa.accept();
  SJSocket s_sc = s_sa.receive();
  Address custAddr = s_sc.receive();
  s_sc.send(dispatchDate);
}
```

Additional features In addition to what was illustrated through the example above, 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/Bica

Gay et al. [2010] 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. The type verification 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. Caldeira and Vasconcelos [2011] is such an implementation.

In this system, communication channels are objects and the usual session types for channels can be translated into object sessions. (Note that this translation is not implemented in Bica; currently, in order to typecheck usage of a session-typed channel, its object session type would have to be given directly.)

As opposed to SJ, this work focuses not on communication channels but on the usage protocols of objects in general and on 'modularity'. Modularity in this context means that a method can implement part of the protocol for some object, then set this object aside and return to the caller. Later in the program, another method will get back to the object and complete its protocol.

This is not possible with a system like SJ which has usage protocols only for channels, since it implies that the methods of the enclosing object must themselves be called in the proper order. In SJ, a channel can only be disposed of by terminating its session type completely or passing it as an argument to a method call.

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 Customer class could 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 only branching:

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

By applying the translation we then obtain the following object session type, which will be the type of the channel endpoint object once the connection is established:

```
AcceptOrderEndpoint = {
linkthis receive() :
  <
    QUERY: {String receive() : {Null send(Double) : AcceptOrderEndpoint}};
    ACCEPT: {Address receive() : {Date receive() : {}}};
    REJECT: {}
  >
}
```

The curly braces 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 examinate the value returned by the last method call in order to know how to continue using the object.

AcceptOrderEndpoint is the type the channel object will have when first created. It says: the only method available initially (there is only one thing in the outermost curly braces) is receive(). This method has no argument and its return 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 `receive()`, which will return a String, after which a call to `send` 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 `receive()` is available as well, but its return type is different in this case, as well as the subsequent session type. If the return value is REJECT, no method can be called anymore.

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

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

This gets translated into an object session type as follows (note that the duality between the two endpoint types is not obviously visible anymore):

```
PlaceOrderEndpoint = {
  Null send({QUERY}) : {Null send(String) : {Double receive() : PlaceOrderEndpoint}};
  Null send({ACCEPT}) : {Null send(Address) : {Null send(Date) : {}}};
  Null send({REJECT}) : {}
}
```

In this type, {QUERY}, {ACCEPT} and {REJECT} are singleton types, i. e. particular cases of enumerated types, which can in general be any finite set of labels. Thus the initial set of available methods contains three elements, with the same method name but different (and disjoint) argument types. The meaning is that code using the object can do any of the three method calls and must then follow the subsequent protocol associated to that one.

Note: in this system, a set like {QUERY, ACCEPT} is a valid enumerated type and a supertype of both {QUERY} and {ACCEPT}. The type system would however not allow, given the session type above, a call to `send` with an argument of type {QUERY, ACCEPT}, since it would not be possible to know statically which branch is taken.

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

```
PlaceOrderEndpoint c = agencyAccesspoint.request();
boolean decided = false;
while(!decided) {
  c.send(QUERY);
  c.send(journeyDetails);
  double cost = c.receive();
  //set decided to true or change details and retry
}
if (want to place an order) {
  c.send(ACCEPT);
  c.send(address);
  Date dispatchDate = c.receive();
} else {
  c.send(REJECT);
}
```

where `agencyAccesspoint` has type `<acceptOrder>`. It represents a URL and is supposed to be defined globally, like classes, via an `access` declaration. This declaration contains the URL and the *channel* session type acceptOrder. When `request()` is called, the result is an object whose type is the object translation of the *dual* of the channel session type, so here the type is PlaceOrderEndpoint.

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), i. e. AcceptOrderEndpoint.

Delegation The language described in the paper allows delegation, like MOOSE and SJ. It is however not implemented in Bica. We show how the code would be written nevertheless.

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 a call to `receive()` is possible and it will return an object with session 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 show how the customer class could be structured in a way that takes advantage of modularity.

```
class Customer {
// Session declaration
  session Init
  where Init = {Null connect(<acceptOrder>) : {Double getPrice(String) : S}}
  and S = {
    Double getPrice(String) : S;
    Date placeOrder(Address) : Init;
    Null cancel() : Init;
  }

  c; //this declares a field named c

//then the method bodies are declared
  connect(agency) {
    c = agency.request();
  }
  getPrice(journeyDetails) {
    c.send(QUERY);
    c.send(journeyDetails);
    return(c.receive());
  }
  placeOrder(address) {
    c.send(ACCEPT);
    c.send(address);
    return(c.receive());
  }
  cancel() {
    c.send(REJECT);
  }
}
```

Note that the field declaration does not have a type, because the field's type is allowed to change over time; the method declarations do not have types either because their types are in the session type of the class.

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 type Null because it is not initialized. Then the first method called must be `connect` with an argument type of `<acceptOrder>`. The body of `connect` is typable in this context and changes the type of the field to PlaceOrderEndpoint (i. e. the translated dual type of acceptOrder).

Then the session type of Client 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, etc.

Mool Campos and Vasconcelos [2010] extends modular session types for objects in two ways. (1) Bica treats communication 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) Bica 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 qualiers 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>;
  // the class fields
  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 return 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 unrectricted 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 on the object, the typestate of the object may change, thereby dynamically changing the object's set of permitted operations. A *typestate precondition* must hold for an operation to be applicable, and a *typestate postcondition* reflects the possible typestates 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. 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.

Fugue was the first modular verification system for specifying and statically checking typestate properties for .Net-based programs, and adapts Strom and Yemini's typestates to object-oriented programs. The typestate system of Fugue is presented by DeLine and Fähndrich [2004]. In Fugue, a typestate is an abstraction over the concrete state of an object; i.e., a typestate is 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 presented in Fähndrich and DeLine [2002] 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.

Typestate-oriented programming integrates typestates directly into the language design instead of integrating typestates with the features of existing languages. Aldrich et al. [2009] 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. [2011b,a]. In contrast to Fugue, 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`. 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:

```
state Customer {
  Agency a;
}
state Ordering case of Customer {

  void init () {
    Double d = getPrice("string"); ... ;
    if (good_offer("string",d)){ this<-Accepting(a,"string"); a.accept("string")
      } else { this<-Rejecting; a.reject(); }
  }
}
state Accepting case of Customer {
  String s;
  ...
}
state Rejecting case of Customer {...}
```

If the client is in typestate `Ordering`, it can start the session by calling the `init` method. Note how the successful `"string"` argument is passed to the `Accepting` state. Similarly, the `Agency` could consist of the typestates `OrderSession`, `Accept`, and `Reject`.

```
state Agency {
  Service service;
}
state OrderSession case of Agency {
  Double getPrice(String s){...} // calculate the price

  Date accept(Address a){
```

```
    this<-Accept;
    Session s = service.createSession();
    return s.deliveryAddress(a);
  }

  Void reject(){ this<-Reject; }
}
state Accept case of Agency {
  Double getPrice(String s){ this<-OrderSession; ...} // calculate the price
}
state Reject case of Agency {
  Double getPrice(String s){ this<-OrderSession; ...} // calculate the price
}
```

Here, we see how external choice is captured by different methods and internal choice by a conditional. 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 by 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){...}
}
```

Recent work on gradual typestates by Wolff et al. [2011] use access permissions, similar to Fugue, as aliasing annotations to control references. References with `full` permissions have exclusive write access, references with `shared` permissions have write access, and references with `pure` permissions have only read access. This approach allows the number of dynamic checks to be reduced at the overhead of adding annotations to the program, while retaining a modular static analysis technique.

3 Other Paradigms

3.1 Functional Languages

The integration of sessions and of session types in functional languages poses two main challenges. The first one concerns both lazy and strict functional languages and regards the fact that, by their own definition, session types describe entities (channel endpoints) whose capabilities may change after each usage. This feature is at odds with conventional types used in functional languages, which 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 evaluation order of expressions is difficult to control in lazy functional languages (in particular, Haskell). 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 extensively investigated for an ML-like language by Vasconcelos et al. [2006], Gay and Vasconcelos [2010]. Vasconcelos et al. [2006] take a conventional approach with respect to session types and extend function types to enable the description of the *effect* of a function on the channel it uses, in a way that resembles effect types. For example, the agency process can be modeled thus:

```
agency :: ⟨Agency⟩ᵃ → Unit
agency agencyAccess = sell (accept agencyAccess)

sell :: s : Agency; Chan s → Unit; s : End
sell s =
  case s {
    QUERY   ⇒ let journeyDetails = receive s in
              send (cost journeyDetails) on s
              sell s
    ACCEPT  ⇒ let address = receive s in
              send (date journeyDetails) on s
    REJECT  ⇒ ()
  }
```

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 behavior of the agency by reading and writing messages on the private session `s` by means of the **`send`** and **`receive`** primitives. The **`case`** construct is also related to communication: it waits for a label (one of `QUERY`, `ACCEPT`, or `REJECT` in the example) and evaluates the code that follows the actual label received from the session.

The type of `sell` specifies that the function accepts a channel as argument, 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 type alias for a conventional session type:

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

While analyzing the body of `sell`, the type checker tracks each occurrence of the channel `s` and verifies that it is consistent with its type, which is updated as the analysis works through the code from top to bottom.

In a subsequent work, Gay and Vasconcelos [2010] take a rather different approach whereby channels are treated as truly linear resources that must be used exactly once. The idea is that a function that takes a channel as argument *consumes* the channel, which is not available for further operations. However, the function may *produce* a continuation of the same channel, which can have a possibly different type. Following this style, the `sell` function above 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
  }
```

Note that the **`case`** construct now expects functions on the rhs of ⇒'s, which are applied with the continuation of `s` (next to the **`case`** keyword) after the label has been received. Also, the **`receive`** and **`send`** constants are given the following type schemas:

```
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. These types explain the subsequent *rebindings* of the channel `s` in the code above. In fact, each occurrence of `s` is virtually a different channel that either is used exactly once, or it is returned by the function. We say "virtually" because in practice there is just one session channel which is re-used over and over again after each input/output operation. The rebinding is thus meaningful only at the type level, allowing each occurrence of `s` to be associated with a possibly different type. Interestingly, these re-bindings are reminishent of the compilation scheme of sessions into pure π-calculus channels [Dardha et al., 2012] as well as of monadic handling of state in pure functional languages such as Haskell, whereby the "state" `s` is threaded in a strictly sequential way and the rebinding boilerplate code is implicit and hidden within the monad definition (a monadic treatment of sessions for Haskell is indeed feasible and is described by Pucella and Tov [2008], which we discuss below).

This approach conceals a problem caused by the fact that the type of a function provides no information whatsoever on the possible *free* objects that a function may store 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. 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
```

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 communication topologies that leave some channels unreachable.

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 access to a session. The use of a monad dedicated to session interactions also solves the *aliasing problem*. Unlike [Vasconcelos et al., 2006, Gay and Vasconcelos, 2010, Bono et al., 2013], which *extend* the type system of an existing language and add support for linear types, Neubauer and Thiemann [2004], Pucella and Tov [2008] encode sessions using the features of Haskell's type system, which makes no provision for linear values. 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.

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 above, except that there is no explicit argument denoting the channel on which these operations act. The actual channel is hidden in the definition of the `Session` monad, which 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 types which are cumbersome and error-prone to write explicitly. Fortunately, it is possible to take advantage of Haskell's type inference for inferring them in most cases. A more advanced session type inference technique is described by Imai et al. [2010].

Future work Overall the achievements described in the aforementioned works suggest that a good integration of behavioral types in higher-order languages is indeed feasible. It should be remarked, however, that the behavioral types taken into account so far are solely aimed at guaranteeing basic safety properties, and that the enforcement of stronger properties (such as liveness guarantees) usually requires the embedding of more precise information into types. This embedding poses little technical problems at the foundation level, because of the close correspondence between the structure of types and that of processes adhereing to them. However, there is evidence suggesting that the same embedding is not trivial when higher-order languages are considered. This calls for further investigations to maintain the research done at the level of programming languages aligned with that on the foundations of behavioral types.

3.2 High-performance computing

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

MPI programs adhere to the Single Program, Multiple Data paradigm, in which a single program specifies the behaviour 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. Point-to-point communication specifies the interaction between two different processes: a sender and a receiver. The communication can be characterised 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 arises 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 communication allows for the overlapping of computation and communication.

Collective operations are executed synchronously by all (or a group of) processes. These operations include, for instance, the ability to broadcast a buffer for all (or a group) of processes, the capability to scatter or gather a buffer amongst processes, and reducing operations on values from all (or a group) of processes. Collective operations facilitate the writing of complex behaviours and give the opportunity for the MPI implementation to optimise 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 behaviour, MPI implementations exploit the fact that a collective operation is taking place in order to optimise the way communication is handled.

One-sided communication allows a process to remotely access the memory of another process (RMA) for getting and putting values directly on the other's process memory. It differs from the previous modes of communication because the process issuing the request may do it without the collaboration of the other involved process, as in point-to-point and collective communications.

High-performance computing applications can exhibit complex message passing behaviours 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 behaviours. It is, in fact, very easy to write an MPI application that deadlocks or enters into race conditions just by following a wrong communication protocol. For that matter, behavioural types can be of great help, since they can capture the global communication protocol and are able to enforce this behaviour on the program.

Several proposals to discipline communication in high-performance computing have been put forward Ng et al. [2012a, 2011], Honda et al. [2012], Ng et al. [2012c], Marques et al. [2013b], based on the theory of multi-party sessions types. The approach starts by describing a global protocol of the application using a protocol description language, for instance Scribble Honda et al. [2011]. Then, the global protocol is projected to endpoint protocols to be followed by each participant. 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.

Example

The Customer-Agency running example protocol can described in Scribble as follows.

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

A possible implementation sketch in C using MPI primitives (C+MPI) can be as follows.

```
#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 behaviour of both participants at the same time, in the style of single program, multiple data. It starts by initialising the MPI library and by getting the process rank. The rank is an integer that uniquely represents each process. In the present case, the process ranked 0 is the customer, whereas the process ranked 1 is the travel agency. The behaviour 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 use 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.

3.2.1 Session Java

Session Java (SJ) is an extension of Java with session types, supporting statically safe distributed programming by message passing Ng et al. [2011]. The goal of studying distributed programming with session types at a higher level of abstraction is to disentangle communication from computation, as it is the case of C+MPI programs, and to ensure communication correctness (in terms of communication safety and progress) of well-typed programs, as well as to increase productivity and performance.

Session Java uses a server-client programming model, since SJ is based on binary session types. SJ, as a language, and its static type checking approach ensures the compatibility between any two communicating processes. Parallel programming with SJ extends binary sessions to multiple inter-connected binary sessions in parallel. The verification of SJ parallel program therefore requires two component: (1) correctness of each binary sessions, and (2) correctness of the network topology which defines the connection between the individual binary sessions, using "outwhile" and "inwhile" synchronised iteration primitives.

An SJ program consists of a collection of SJ classes, one for each type of process to be deployed. Processes differ by their position in the network topology and by their role in the coordination of the parallel algorithm as a whole. To complete the application there is a configuration file that describes how to connect the process classes. The deployment workflow is describes as follows: (1) SJ classes are compiled into Java standard bytecode using the SJ compiler, which checks the correct implementation of each binary session; (2) the topology verifier checks the topology declared in the configuration file, which in conjunction with step (1) prevents global deadlocks; (3) the verified, compiled files are deployed in the cluster; and (4) program execution makes use of the `ConfigLoader` utility, from the SJ library, to establish sessions with their assigned neighbours in the configuration file, ensuring safe execution of the parallel program.

Complex protocol interactions, like iteration and branching, are coordinated by *active* and *passive* actions at each side of the session. The *master* process decides whether to continue the session iteration using **`outwhile(condition)`**, or selects a branch using **`outbranch(label)`**, whereas the *worker* processes passively follow the iteration or the selected branch decision using **`inwhile`** and **`inbranch`** primitives and proceed accordingly. In more detail, for iterations, two methods are available: *local* and *communicating iterations*. Local iterations is a standard statement, such as `while`-statements, with session operations occurring inside. Communication iterations are a distributed version of loops, where, at each iteration, the loop condition is computed by the process calling **`outwhile`** and is communicated to processes calling **`inwhile`**. The while loop is designed to support multicast, so that a single **`outwhile`** can control multiple processes. This pattern is useful in a number of parallel iterative algorithms, which the loop continues until certain conditions (e.g., convergence) are reached and cannot be determined statically. As for branching, different branches may have different communication behaviours, and the deciding participant needs to inform the other participant which branch is chosen. The passive participant will react accordingly.

The Customer-Agency example written in Session Java

The protocol description and implementation of our running example is presented in Section 2.2.1. The increase in clarity is evident when compared with the C+MPI program presented in the previous section. Thereby, productivity gains are clear. Besides that, SJ offers statically guarantees that well-typed programs are free from deadlocks and that communication is safe. Further work includes the support for more flexible topologies and native compilation for efficiency gains. However, Session Java consistently outperforms MPJ Express,[1] a Java implementation of the MPI standard, a performance competitor with C-based MPICH2 Shafi et al. [2009].

[1] `http://mpj-express.org`

A final aspect to notice is that most programs in the HPC community use collective operations. Building on that, programs are able to maintain a shared state that allows them to decide collectively, based on this state. This happens without having to follow a master-worker pattern, avoiding additional communications for synchronising iterative and branching computations, as seen in SJ.

3.2.2 Session C

Session C Ng et al. [2012a] 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, 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 optimised 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, since it can handle directly multiparty communications—SJ originally treats only binary sessions. To guarantee deadlock freedom and global progress for multiparty sessions, SJ depends on an external tool, as discussed in the previous session. Session C also offers a significant speed-up (60%) compared to SJ as well as MPI for Java.

A Session C application is developed in a top-down approach through four stages. First, a programmer designs a global protocol using Scribble (as shown in Section 3.2). An application is compose by individual programs that implement each participant behaviour. This approach contrasts with that of single program, multiple data put forward by MPI. Second, a projection algorithm takes the global type and generates endpoint protocols, extracting only the interactions that involve each particular participant. Then, in the third stage, a protocol can be refined, meaning that the programmer may write a program that differs from the original protocol up to the reordering of asynchronous messages Mostrous [2009], Mostrous et al. [2009] for minimising the waiting time. The fourth step checks the conformance of the refined C program with a subtype of the endpoint projection.

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 sketch in Session C as follows. Here 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);
```

```
}
```

Follows the code for the travel agency.

```
#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)));
    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 behaviour 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 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.

3.2.3 Deductive Verification of C+MPI programs

This approach directly verifies C+MPI programs against session types Honda et al. [2012], Marques et al. [2013b], in contrast with Session C and Session Java where programmers use a particular library of communication operations. This approach is also founded on the theory of Multiparty Session Types.

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 contracts 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 9.

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 behaviours 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.

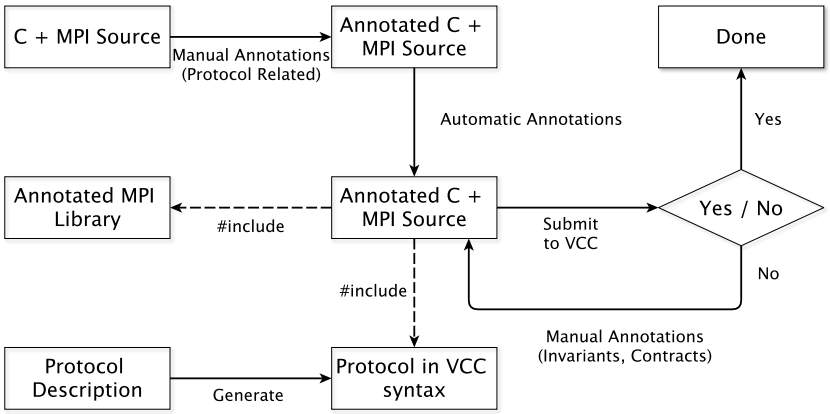


Figure 9: Approach for verifying C+MPI programs

The verification process checks the program from MPI initialisation (call to `MPI_Init`) to shutdown (`MPI_Finalize`). There is the need to add state and behaviour 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 in 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 sketch 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 interesting examples include collective operations and foreach loops, but were omitted since they are not part of our running example.

Open issues and future work

There were plans to explore more complex topologies that can be supported by Session Java and its topology verifier. Due to the similarity of the approach with Session C and that Session Java only supports binary session types, which limits the usefulness in supporting collective operations in parallel programming, efforts of further extending the session-based approach is now focussed on Session C.

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 comprehensiveness and that it is standardised, and to a lesser extent, 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 indices. 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 also 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.

3.3 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 section 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 Jason is very challenging, since like the majority of logic-based languages, Jason is statically untyped; for this reason any non trivial static analysis turns out to be very difficult and algorithmically intractable, unless one implements a non downward-compatible extension of the language by introducing explicit type annotations.

We have therefore opted for investigating dynamic verification of protocol conformance for Jason; even though this choice implies less guarantees in comparison with the static approach,

dynamic verification has the advantage that more expressive formalisms can be used, since the undecidability issues typical of static analysis are a less serious concern.[2]

Although a more expressive formalism makes specifications more concise and readable, verification becomes more challenging, because, in order to be effective, dynamic checks need to be efficient. 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.

In this section we introduce the notion of global type presented in previous work Ancona et al. [2012, 2013a,c], Mascardi and Ancona [2013].

Global types can be easily represented as cyclic Prolog terms, and a mechanism for verifying that a sequence of messages complies to 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 agents in the MAS complies to the interaction protocol specified by a global type in the formalism described in this section.

Interactions. An interaction occurs between two agents and is a 4-tuple consisting of two agent identifiers (the sender and the receiver of the message), the performative expressed in some agent communication language the agents agree upon, such as FIPA-ACL Foundation for Intelligent Physical Agents [2002] 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 the interaction `ca(seller, buyer, tell, price(pasta,10))` specifies that agent `seller tells` agent `buyer` that he 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.

The set of interactions is denoted by $\mathcal{A}$ throughout this section.

Interaction types. In the specification of the 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 who 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.

[2]Most decision problems are already undecidable for context-free languages, whereas the formalism adopted here is strictly more expressive than context-free grammars.

Example

The Customer-Agency example described in [CDLPRIMER] can be modeled by global types in the following way.

$$CustomerAgency = Sell \cdot AcceptOrReject$$

$$Sell = query^0 : propose^0 : (Sell + \lambda)$$

$$AcceptOrReject = (Accept + Reject)$$

$$Accept = confirm^0 : sendAddress^0 : forwardAddress^0 : sendDate^0 : forwardDate^0 : \lambda$$

$$Reject = reject^0 : \lambda$$

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 must 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

$$Sell = query^0 : propose^0 : (Sell + \lambda)$$

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 address where delivering 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.

Note: the 0 superscript of all interaction types can be ignored for this example; it will be introduced and explained in the "specific examples" subsection below.

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 that 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.

Peculiarities

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 is allowed to terminate, but 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 one side, static verification ensurers strongest guarantees on the correct behavior of a MAS, on the other side, dynamic verification allows the adoption of much more expressive languages. For instance, since global types are recursive and support concatenation, context-free languages can be specified[3] 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 constraing 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 specified with global types. For instance, the typical example of non context-free language[4] $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ can be easily specified (see the next section).

Specific examples

We consider the ABP, in the version defined by Deniélou and Yoshida 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.

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

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

- 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 not very 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.

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 of 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 first Alice 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:

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

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.

Further reading

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 a global perspective is taken. 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 in 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.

With the Deniélou and Yoshida global types 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 decribed 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, comes from Chen et al. [2011a]. There, a formal model of run-time safety enforcement for largescale, 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 material related to global types for MASs, including papers, the code of working prototypes, and implemented examples, can be found here: `http://www.disi.unige.it/person/MascardiV/Software/globalTypes.html`. The Jason monitor is discussed in Ancona et al. [2012], whereas Mascardi and Ancona [2013] discusses 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 in Ancona et al. [2013b].

```
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        ``produce 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    }
15  }
```

Figure 10: Agency.

4 Singularity OS

Singularity OS Hunt et al. [2005], Fähndrich et al. [2006] is the prototype of a dependable operating system where software-isolated processes (SIPs) run in the same address space. Process interaction occurs solely through the exchange of messages over asynchronous, FIFO channels and the communication overhead is tamed by copyless message passing: only *pointers* to messages are physically transferred from one process to another. Static analysis guarantees *process isolation*, namely that every process can only access memory it owns exclusively. The Singularity OS implementation can be found at `http://singularity.codeplex.com/`.

Sing$^{\#}$ is the programming language specifically designed for the development of programs that run in Singularity OS. We take now a closer look at Sing$^{\#}$ by means of the Costumer-Agency example.

It is useful to know that Singularity channels consist of pairs of related *endpoints*, called the *peers* of the channel. Messages sent over one peer are received from the other peer, and vice versa. Each peer is associated with a FIFO buffer containing the messages sent to that peer that have not been received yet. Therefore, communication is asynchronous (send operations are non-blocking) and process synchronization must be explicitly implemented by means of suitable handshaking protocols. Channel communication is, in fact, governed by statically verified *channel contracts* that describe messages, message argument types, and valid message interaction sequences as finite state machines similar to session types.

The pseudocode snippet in Figure 10 defines a function `Agency` that encodes the behaviour of the process Agency. The function accepts two arguments: a `c` endpoint representing one peer of the channel used as the session channel to interact with the Customer (which the other peer endpoint belongs to); a `d` endopoint representing one peer of the channel exploited to delegate `c` to the Service at the appropriate moment. 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 is expected and the tag of the message. If a request of details comes (message `Query` on line 4), a proposal is sent (line 6), and the function `Agency` is invoked recursively (line 7), so that the negotiation might continue. In the case a `Reject` message is received (line 8), both endpoints are closed, as the negotiation failed. If a `Accept` 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 unware of the change of interlocutor. The operational semantics of the processes Customer and Service,

```
1   void Customer([Claims] exp<C:START> in ExHeap c,
2                 String destination, String address) {
3      c.Query(destination);
4      switch receive {
5       case c.Price(double p):
6         if (``p not ok and negotiate'')
7          Customer(c, destination, address);
8         else
9           if (``p not ok and reject''){
10           c.Reject();
11           c.Close();
12         }
13         else {
14          c.Accept();
15          c.Address(address);
16          switch receive {
17           case c.Date(String d):
18            c.Close();
19          }
20       }
21     }
22  }
```

Figure 11: Customer.

```
1   void Service([Claims] imp<D:START> in ExHeap d) {
2      switch receive {
3        case d.Delegate(<C:ADDRESS> in ExHeap x):
4         switch receive {
5          case x.Address(String a):
6           ``produce a date da for a''
7           x.Date(da);
8           x.Close();
9         }
10        d.Close();
11     }
12  }
```

Figure 12: Service.

encoded as functions `Customer` and `Service` and shown in Figure 11 and Figure 12, should be self-explanatory. We illustrate now the meaning of the type annotations and their relevance with respect to static analysis. The `in ExHeap` annotation state that a name denotes a pointer to an object allocated on the exchange heap. 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. 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 aim, the designers of Sing# have consequently devised channel contracts describing the allowed communication patterns 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* connected by *transitions*. Each message specification begins with the `message` keyword and is followed by the *tag* of the message and the type of its arguments. The state of the contract determines the state in which the endpoint associated with the contract is and this, in turn, determines which messages can be sent/received. Communication errors are avoided by associating the two peers of a channel with types that are complementary, in that they specify complementary actions. This is achieved in Sing# with the `exp<C:`*s*`>` and `imp<C:`*s*`>` type constructors that, given a contract `C` and a state *s* of `C`, respectively denote the so-called *exporting* and *importing* views of `C` when in state *s*. It is useful to think of the exporting view as of the type of the *provider* of the behavior specified in the contract, and of the importing view as of the type of the *consumer* of the behavior specified in the contract.

Going back to the Costumer-Agency example, note that Service waits for an endpoint value in `x` (Figure 12, 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 seems plausible to formalize Sing# using a process calculus equipped with a suitable session type system for endpoint types. There are, in fact, clear analogies between contracts and endpoint types: the contract describes an interaction between two processes in terms of states and transitions, with a bias towards one of the two processes; the endpoint type describes the behavior of a single process involved in the interaction.

An encoding of the example Costumer-Agency in the style of Bono et al. [2011] may look like as:

$$
\begin{array}{lcl}
\text{INTERACTION}(de, ad, pr, da) & = & \texttt{open}(c : T, c' : \overline{T}).(\text{CUSTOMER}(c, de, ad) | \\
& & \texttt{open}(d : D, d' : \overline{D}).(\text{AGENCY}(c', d, pr) | \\
& & \text{SERVICE}(d', da))) \\
\text{CUSTOMER}(c, de, ad) & = & c!\texttt{Query}(de).c?\texttt{Price}(p : double). \\
& & \texttt{rec}\ X.(c!\texttt{Query}(de).c?\texttt{Price}(p : double).X \oplus \\
& & c!\texttt{Reject}().\texttt{close}(c) \oplus \\
& & c!\texttt{Accept}().c!\texttt{Address}(ad).c?\texttt{Date}(da : String).\texttt{close}(c)) \\
\text{AGENCY}(c', d, pr) & = & \texttt{rec}\ X.(c'?\texttt{Query}(x : String).c'!\texttt{Price}(p).X + \\
& & c'?\texttt{Reject}().\texttt{close}(c').\texttt{close}(d) + \\
& & c'?\texttt{Accept}().d!\texttt{Deleg}(c').\texttt{close}(d)) \\
\text{SERVICE}(d', da) & = & d'?\texttt{Deleg}(x : \overline{S}).x?\texttt{Address}(a : String).x!\texttt{Date}(da).\texttt{close}(x).\texttt{close}(d')
\end{array}
$$

The operational semantics of the construct `open` present in the process INTERACTION corresponds to the creation of a channel by allocating the peer endpoints in the heap. The endpoint types appearing in the code are as follows:

$$
\begin{array}{l}
T = \texttt{rec}\ \alpha.\ (!\texttt{Query}(String).?\texttt{Price}(double).\alpha \oplus \\
\qquad !\texttt{Accept}().!\texttt{Address}(String).?\texttt{Date}(String).\texttt{end} \oplus \\
\qquad !\texttt{Reject}().\texttt{end})
\end{array}
$$

$$
D = !\texttt{Deleg}(S).\texttt{end}
$$

$$
S = !\texttt{Address}(String).?\texttt{Date}(String).\texttt{end}
$$

Note that S is a suffix of one of the internal choices of T and this is the parallel of requiring `x` of type `<C:ADDRESS> in ExHeap` on line 3 of the function `Service` (Figure 12).

A major complication of the copyless paradigm derives from the fact that communicated objects are not copied from the sender to the receiver, but rather pointers to allocated objects are passed around. This can easily invalidate the ownership invariant if special attention is not payed to whom is entitled to access which objects. Any chosen type discipline, then, should control the *ownership* of allocated objects, whereby at any given time every allocated object is owned by one (and only one) process. Whenever (the pointer to) an allocated object is sent as a message, its ownership is also transferred from the sender to the receiver. There are two possible cases for ownership of parameters: either the ownership is given back to the caller (no annotation), or it is retained by the callee (annotation `[Claims]`) after the execution. In the example of Figure 10, 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 away endpoint `c` (transferring the ownership to the receiver) and closes endpoint `d` in the case of acceptance (lines 12 and 13).

One could hope that, by imposing a *linear* usage on entities, the problems regarding the ownership of heap-allocated objects would be easily solved. In practice, things are a little more involved than this because, somewhat surprisingly, linearity alone is *too weak* to guarantee the absence of *memory leaks*, which occur when every reference to an heap-allocated object is lost. We illustrate this issue through a simple example. Consider the function:

```
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 exchange heap, sends endpoint `f` as an `Arg`-tagged message on `e`, and closes `e`. The `[Claims]` annotations in the function header are motivated by the fact that one of the two arguments is sent away in a message, while the other is properly deallocated within the function. Yet, this function may produce a leak if `e` and `f`

are the peer endpoints of the same channel. If this is the case, only the `e` endpoint is properly deallocated while every reference to `f` is lost and will never be deallocated. Note that the `leak` function behaves correctly with respect to the Sing# contract

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

whose only apparent anomaly is the implicit recursion in the type of the argument of the `Arg` message, which refers to the contract `C` being defined.

An encoding of this example in the calculus 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, S are dual 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 problem does not lie in the implicit recursion in the type of the argument of the `Arg` message, but in the fact that LEAK creates a memory loop: at the end of the execution, the endpoint f is present in its own FIFO queue, as the argument of a message that will be never read. Therefore, loops of this nature should be avoided: the intuition is that such a message queue containing a loop has an infinite "depth". In fact, the idea introduced in Bono et al. [2011] is to define a notion of *weight* for endpoint types which roughly gives the "depth" of the message queues in the endpoints having those types and to restrict endpoint types to those having finite weight.

Remarkably, the `leak` function is ill typed also in Sing# Fähndrich et al. [2006], although the motivations for considering `leak` dangerous come from the implementation details of ownership transfer rather than from the memory leaks that `leak` can produce. Having pinpointed the actual reason why the function `leak` is faulty is the main contribution of the formalization of Sing# in Bono et al. [2011].

The interested reader can consult the following sources:

- Fähndrich et al. [2006] report on the language, verification, and run-time system features that make messages practical as the sole means of communication between processes in the Singularity operating system. They show that using advanced programming language and verification techniques, it is possible to provide and enforce strong system-wide invariants that enable efficient communication and low-overhead software-based process isolation. An (informal) overview of Singularity OS specifications is in Hunt et al. [2005].

- Bono et al. [2011], Bono and Padovani [2012] 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 inadequate, because it leaves room for scenarios where well-typed processes leak significant amounts of memory.

- 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 mentioned earlier. Notably, when polymorphic endpoint types are allowed, a simple (polymorphic) variant of the process LEAK can be typed without resorting to recursive types. The notion of weight extends smoothly to type variables: when α occurs in a constraint $\alpha \leqslant t$, we estimate the weight of α to be the same as the weight of t.

- Stengel and Bultan [2009] show that they are implementable without deadlocks if they are *deterministic* and *autonomous*. The first condition requires that there cannot be two transitions that differ only for the target state. The autonomous condition requires that every two transitions departing from the same state are either two sends or two receives. These conditions make it possible to split contracts into pairs of dual session types, and to fit existing session type theories in our setting in such a natural way.

- 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.

- Jakšić and Padovani [2012, 2013] extend [Bono et al., 2011, Bono and Padovani, 2012] with exceptions. The semantics of processes draws inspiration from software transactional memories: a transaction is a process that is meant to accomplish some exchange of messages and that should either be executed completely, or should have no observable effect if aborted by an exception.

- Bono et al. [2013] extend the technique presented in Bono et al. [2011], Bono and Padovani [2012] for detecting memory leaks to a language with first-class functions. These results are part of the stream of work on functional languages (see Section 3.1). The technique based on weights mentioned above does not work directly in a language with first-class functions. The problem is that arrow types only tell us what a function accepts and produces, but not which other (heap-allocated) values the function may use, while this information is essential for determining the weight of a linear arrow type. The devised solution is to decorate a linear arrow type with a weight which is an upper bound for the weights of the types of all endpoints occurring in the function body. As before, the weight represents an approximation of the length of a chain of pointers in the program heap, therefore it is possible to send a function value over an endpoint only if its weight is bounded.

5 Web Services

5.1 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: executable languages which perform activities by means of local computations combined with invokations to other services. In this context, behavioural abstractions, which can be seen as an analogous of behavioural types extracted from a program written in an orchestration language (see, e.g., 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 behavioural contracts (see, e.g., Fournet et al. [2004], Bravetti and Zavattaro [2007, 2008]). 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] or succesfully terminates Bravetti and Zavattaro [2007, 2008]. Session types make it possible to extract such behavioural descriptions (in the form of types) from the actual service code (type inference) or to directly check that service code conforms to a given behavioural description (type checking). Type checking crucially relies on a sub-typing relation between session types that is defined to be the coarsest possible that preserves the desired termination properties, so to be as permissive as possible when typing code (we will discuss this with examples in Section 5.2.2).

In order to be able to perform this kind of checks, it is necessary for the services to expose in their interface also the description of their expected behaviour. In general, a service interface description language can expose both *static* and *dynamic* information. 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 operations in order to correctly complete a session of interaction. The WSDL, which is the standard Web Services interface description language is basically concerned just with static information.

In the following we will deal with languages for representing concrete and abstract service orchestrations. The idea would be to then exploit such abstract representations to enrich information provided in WSDL.

5.2 Related Approaches

Concerning inclusion of abstract service descriptions in WSDL, for instance SAWSDL [Kopecky et al., Nov.-Dec.] provides a mechanism for adding semantic annotations in WSDL. It would be interesting to see whether an extension to SAWSDL with behavioral information is possible.

In addition, other kind of checks can be obtained by enriching information included in WSDL descriptions. For instance, Allison et al. [2012] deals 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). It is a specific negotiation case (for privacy purposes), but shows the process of negotiation that is relevant to BETTY. Specifically, the policy languages employed in such negotiations are relevant to WG3 (e.g., eXtensible Access Control Markup Language - XACML).

5.2.1 Concrete Languages for Service Compositions

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 in [Montesi et al., 2007] as an orchestration language for Web Services alternative to the standard language WS-BPEL: one of the main advantages is that Jolie does not have an hard to read XML syntax, but on the contrary has a more programmer-friendly syntax similar to C/Java. Jolie has been subsequently extended 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, in 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, 2013, 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:

```
1  main
2  {
3    start@Agency()(m.sid);
4    satisfied = false;
5    for(i = 0, !satisfied && i < 5, i++) {
6      showInputDialog@SwingUI("Product Name")(m.product);
7      askPrice@Agency(m)(price);
8      showYesNoQuestionDialog@SwingUI(string(price))(answer);
9      satisfied = !bool(answer)
10   };
11   showYesNoQuestionDialog@SwingUI
12     ("Buy " + m.product + " for " + price + "?")(answer);
13   satisfied = !bool(answer);
14   if (satisfied) {
15     accept@Agency(m);
16     order@Agency(m)(date)
17   } else {
18     reject@Agency(m)
19   }
20 }
```

Above, the customer program starts by sending a message for operation `start` to `Agency`, an external reference pointing to the Jolie program 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. We store such session identifier in the variable `m.sid`, and we will use the data structure `m` in the rest of the code to refer to the correct session inside the agency. Lines 5–10 implement the loop of the use case; here, we allow the customer to request the price for a product at most 5 times or until she is satisfied. In Line 6, we use an external service `SwingUI` (provided by the Jolie standard library) for interacting with the user and ask her which product she desires to buy. In Line 7, we ask 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, we ask the user whether she wishes to proceed with the purchase of the last selected product. If so, in Lines 15–16 we send a message for the `accept` operation offered by the agency and we place the order using another message; when placing the order, we expect to receive the expected delivery date for the product as a reply. Otherwise, if the user does not wish to proceed, we invoke the agency on operation `reject` and the session terminates.

We now show 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 1, 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 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.

5.2.2 Abstract Languages for Service Compositions

The orchestration language WS-BPEL can be used to describe so-called abstract processes, that is behavioural descriptions which include unspecified parts, hence may represent just the externally visible communicating behaviour 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 succesfully interact with it.

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

The achieved abstraction, expressed in an 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), e.g. stuck freedom Fournet et al. [2004], termination (under fairness assumption) Bravetti and Zavattaro [2007, 2008], ... In particular, such analysis is often carried out by resorting to more low level semantical descriptions of service behaviors (essentially labeled transition systems) called *behavioural contracts*. One of the most important aspect of the service contract technology is considered to be *correctness of composition*: 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 behaviour.

We exemplify abstract process representation by providing the description of the Customer-Agency use case with abstract WS-BPEL. In order to avoid writing (unreadable and long) XML code we adopt the, quite intuitive, notation of BPELscript. The representation of the Customer is the following.

```
while (  !satisfied(price) ) {
  price=askPrice (A);
};
```

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

Notice that functions $stisfied()$ 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 behavioural abstraction of the Jolie code given in the previous section, where the $i < 5$ constraint in the for loop is disregarded.

Notice that, in order to reason about such service abstraction, it is common to resort to a more semantical notation which just expresses the behaviour 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, receive/onMessage on operation a, and internal computations τ.

For example, using a regular expression notation, the Customer service contract is:
$(\overline{askPrice}_A; price)^*; (\overline{accept}_A; \overline{order}_A; date) + \overline{reject}_A$,
while the Agent service contract is:
$(askPrice; \overline{price}_C)^*; (accept; order; \overline{date}_C) + reject$.

Such contracts can be also derived by projection from the choreografical description given in Section 6.1.

An important topic in this regard is service substitutability. It is a fundamental notion in behavioural 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 behaviour, 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' successfully interacts with any environment (set of contracts of other services) successfully interacting with C. As we already mentioned, in the context of session types, where behavioral descriptons are used as types for actual service code, contract refinement corresponds to the definition of sub-typing. It is thus quite immediate to observe that one of the main challanges 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 contracts above for the *Customer* and the *Agency* is a correct service composition: it is both stuck-free Fournet et al. [2004] and always leads to termination of all interacting contracts (assuming fairness) Bravetti and Zavattaro [2007, 2008].

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 behavioural abstraction of the Jolie code for the Customer given in the previous section.

According to the the theory in Bravetti and Zavattaro [2007, 2008] the contract for this service is a refinement of the contract for the previous 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 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 reason for originating a refined contract is that it simply differs for an additional input on the *loan* channel, modelling the possibility for the Agency to receive a loan request.

6 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 behaviour 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]):

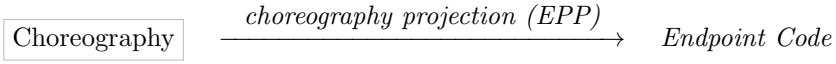
1. Customer -> Agency : *product*;
2. Agency -> Customer : *price*

The choreography above describes the behaviour of two endpoints, Customer and Agency. In Line 1, Customer sends to Agency a *product* name; then, in Line 2, Agency replies to Customer with the *price* of the product she requested.

The following discusses two approaches to developing communication-based software using choreographies. In Section 6.1, 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 formalisation 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 Section 6.2, a choreography is treated as a global specification of an asynchronous communication protocol that is used to verify, by static type checking or dynamically through decentralised run-time monitoring, the conformance of each endpoint process to the protocol.

6.1 Choreography 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 behaviour 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 depict such methodology in the following:



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 used in many theoretical works, e.g., 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 formalisation of WS-CDL is provided by Carbone et al. [2012]. Montesi and Yoshida [2013] present 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.

Currently, the most renown implemented choreography languages are WS-CDL CDL and BPMN BPMN, which do not come with a behavioural 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.

```
program customer_agency;

protocol PurchaseProtocol {
  Customer -> Agency: askPrice(string);
  CheckPrice
}

protocol CheckPrice {
  Agency -> Customer: price(int);
  Customer -> Agency: {
  askPrice(string);
    CheckPrice,
  accept(void);
    Customer -> Agency: order(string);
    Agency -> Customer: date(string),
  reject(void)
  }
}

public agency_url: PurchaseProtocol

define checkPrice(c,a)(s[CheckPrice:c[Customer],a[Agency]])
{
  ask@a(prod,price);
  a.price -> c.price: price(s);
  ask@c(price,satisfied);
  if(satisfied == "Yes")@c {
    ask@c("Confirm?",confirm);
    if (confirm == "Yes")@c {
      c -> a: accept(s);
      c.prod -> a.prod: order(s);
      ask@a(prod,date);
      a.date -> c.date: date(s)
    } else {
      c -> a: reject(s)
    }
  } else {
    ask@c("Product Name",prod);
    c.prod -> a.prod: askPrice(s);
    checkPrice(c,a)(s)
```

```
41    }
42  }
43
44  main
45  {
46    c[Customer] start a[Agency]: agency_url(s);
47    ask@c("Product Name",prod);
48    c.prod -> a.prod: askPrice(s);
49    checkPrice(c,a)(s)
50  }
```

In the program above, we start by defining a protocol for our example, named `PurchaseProtocol`. In Chor, protocols are behavioural types describing the structure of the communication flow between some roles; in this case, our roles are `Customer` and `Agency`. 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 we recur the protocol; (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.

Below the protocol definitions, we have a choreography of our system implementation that follows the previously defined protocols. Procedure `main` is the choreography entry-point of execution. In Line 46, we start a session `s` between a customer and an agency processes respectively called `c` and `a`. The two processes synchronise on the public URL `agency_url`. In Line 47, the customer internally computes (by asking its user through a user interface) the product to buy; then, it asks the agency for the price of the product in Line 48 and the whole system proceeds as defined by procedure `checkPrice`.

Procedure `checkPrice` implements protocol `CheckPrice`. In Line 22, the procedure is declared together with the processes and sessions it uses, respectively `c,a` and `s`. All parameters are behaviourally typed, indicating which protocol each session should implement and which role each process plays in the sessions. In this case, we are declaring that session `s` implements protocol `CheckPrice` using process `c` as the customer and process `a` as the agency. Lines 24–41 follow the structure of protocol `CheckPrice`, where the most notable differences are the usage of concrete data values and the conditional construct `if`.

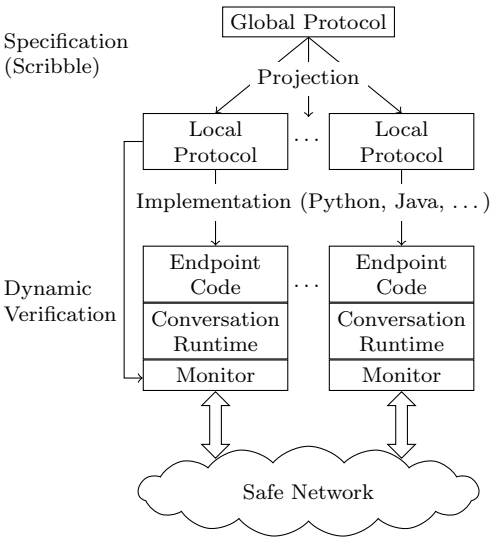
Using Chor, the choreography above can be automatically translated into an executable implementation in the Jolie language development team. In § 5.1, we show how to relate our Chor implementation of the Customer-Agency example to other behavioural analyses in the context of Web Services. Observe that Chor uses multiparty session types, from Honda et al. [2008], as protocol specifications and repetition is thus expressed through recursion. Differently, in § 5.1, we will employ the notation used in Lanese et al. [2008], Bravetti and Zavattaro [2012] for service contracts and will thus refer to the following alternative choreographical representation:

$$(askPrice_{C\to A}; price_{A\to C})^*; ((accept_{C\to A}; order_{C\to A}; date_{A\to C}) + reject_{C\to A})$$

6.2 Scribble

The Scribble project Honda et al. [to appear, 2011], Red Hat JBoss, Team [a] is a collaboration between session types researchers and architects and engineers from industry Savara, Initiative towards the application of session types principles and techniques to current engineering practices. Building on the theory of multiparty session types Honda et al. [2008], Bettini et al. [2008b] (MPST), this ongoing work tackles the challenges of adapting and implementing session types to meet the real-world requirements of our industry partners. 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.




```
1  module TravelAgency;
2
3  type <java> "java.lang.Double" from "rt.jar" as Double;
4  type <java> "java.lang.String" from "rt.jar" as String;
5  type <java> "java.util.Date" from "rt.jar" as Date;
6
7  global protocol BookJourney(role Customer as C,
8        role Agency as A, role Service as S) {
9    rec LOOP: {
10     choice at C {
11       query(journey:String) from C to A;
12       price(Double) from A to C;
13       continue LOOP;
14     } or {
15       choice at C {
16         ACCEPT() from C to A;
17         details(String) from A to S;
18         Address(String) from C to S;
19         (Date) from S to C;
20       } or {
21         REJECT() from C to A;
22         REJECT() from A to S;
23 } } } }
```

Figure 13: (a) The Scribble framework for distributed software development, and (b) a Scribble specification of a global protocol for the Online Travel Agency use case

The Scribble language is a platform-independent description language for the *specification* of asynchronous, multiparty message passing protocols Team [b], Honda et al. [to appear, 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 Red Hat JBoss and Python Initiative 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] the interactions perfomed by the endpoint against the local protocol for its role in the conversation. In addition to internal monitors at the endpoints, Scribble also supports the deployment of external conversation monitors within the network Chen et al. [2011b].

The Scribble framework combines these elements to promote the MPST-based methodology for distributed software development depicted in Figure 13. 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 the research threads motivated by this work.

Scribble resources are available from the project home pages Team [a], Red Hat JBoss.

Online Travel Agency example To demonstrate Scribble as a multiparty session types language, Figure 13 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 from Agency, the exchange of address details and the ticket dispatch date is now conducted between

Customer and a new party, 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 organised 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.
- Lines 7–8 declare the signature of a global protocol called `BookJourney`. This protocol involves three roles, `Customer`, `Agency` and `Service`, aliased as `C`, `A` and `S`, respectively.
- Lines 9–23 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 labelled 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 11–13: 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 16–19), `A` forwards the details to `S` before `C` and `S` exchange `Address` and `Date` messages; otherwise, `A` forwards the `REJECT` to `S` instead.

Tool usage instructions The Scribble tools are available from Team [a]. Taking the Python-based tools as an example:

- Download and extract the Python tool set. Python 2.7.3 or later is required to run the tools.
- The Scribble listing in Figure 13 should be saved in a file `TravelAgency.scr`, matching the Scribble module declaration.
- Running the `scribblec` tool included in the Python distribution:

  ```
  > scribblec TravelAgency.scr
  ```

  will check all the protocols in this file are well-formed (`BookJourney`). If no problems are found, the tool will complete without any feedback.
- Running:

  ```
  > scribblec TravelAgency.scr -project TravelAgency.BookJourney Customer
  ```

  will perform the projection (discussed below) on the `TravelAgency.BookJourney` protocol for the `Customer` role and output the result in a file `TravelAgency_BookJourney_Customer.scr` in the same directory. The output will be as listed in Figure 14.

```
1  module TravelAgency_BookJourney_Customer;
2
3  type <java> "java.lang.Double" from "rt.jar" as Double;
4  type <java> "java.lang.String" from "rt.jar" as String;
5  type <java> "java.util.Date" from "rt.jar" as Date;
6
7  local protocol BookJourney_Customer at Customer
8       (role Customer as C, role Agency as A,
9        role Service as S) {
10   rec LOOP: {
11     choice at C {
12       query(journey:String) to A;
13       price(Double) from A;
14       continue LOOP;
15     } or {
16       choice at C {
17         ACCEPT() to A;
18         Address(String) to S;
19         (Date) from S;
20       } or {
21         REJECT() to A;
22 } } } }
```

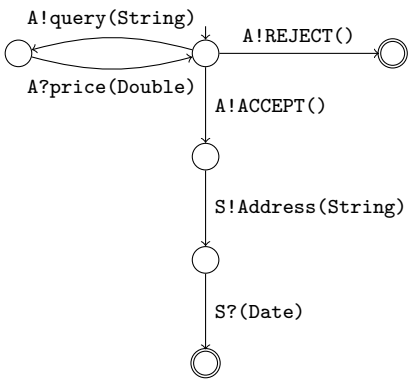


Figure 14: (a) Scribble local protocol for `Customer` projected from the `BookJourney` global protocol, and (b) the FSA generated from the local protocol by the Scribble conversation monitor

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 types 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 formal theory, Scribble features extensions for the practical engineering and maintenance of protocol specifications, such as subprotocol abstraction and parameterised protocols Honda et al. [to appear] (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 development 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. [2012b] 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.

After the specification of the global protocols, the next step of the Scribble framework (Figure 13) 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 (Section 2.2), Bica (Section 2.2) and Sing# (Section 4), 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 Team [b].

Figure 14 lists the local protocol generated by the Scribble tools Team [a] as the projection of the `BookJourney` for the `Customer` role, as identified in the local protocol signature. 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`).

Analogously to 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 to the above session languages, is designed to focus on *dynamic verification* of endpoint behaviour 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 14 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 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. [2011b].

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

The first example demonstrates the abstraction of protocol declarations as *subprotocols*, and the related feature of *parameterised* protocol declarations. Figure 15 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

```
1  global protocol CustomerOptions
2        (role Customer as C, role Agency as A, role Service as S) {
3    choice at C {
4      do GetQuote(C as Customer, A as Agency);
5    } or {
6      choice at C {
7        do Forward<ACCEPT()>(C as X, A as Y, S as Z);
8        do ServiceCall<Address(String), (Date)>(C as Client, S as Server);
9      } or {
10       do Forward<REJECT()>(C as X, A as Y, S as Z);
11 } } }
12
13 global protocol GetQuote(role Customer as C, role Agency as A) {
14   do ServiceCall<query(String, String, String), price(Int)>(C as Client, A as Server);
15   do CustomerOptions(C as Customer, A as Agency);
16 }
17
18 global protocol ServiceCall<sig Arg, sig Res>(role Client as C, role Server as S) {
19   Arg from C to S;
20   Res from S to C;
21 }
22
23 global protocol Forward<sig M>(role X, role Y, role Z) {
24   M from X to Y;
25   M from Y to Z;
26 }
```

Figure 15: Decomposition of the `BookJourney` global protocol using parameterised subprotocols

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

Figure 16: Revision of the `ServiceCall` global protocol with a request cancel interrupt

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 13. It starts with the outer `choice` of `C` to get another quote or move to the quote accept/reject phase. The latter case is given by the inner `choice` whose actions are specified in terms of the `Forward` and `ServiceCall` subprotocols. For example, `do Forward<ACCEPT()>(C as X, A as Y, S as Z)` on line 7 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 request case of the outer 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 parameterisation), 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 Initiative, which could not be directly expressed in the standard MPST formulations. Figure 16 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 synchronisation over 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.

Future work The development of the Scribble framework and its application in real-world use cases is ongoing work. The two main use case projects mentioned in the above are:

Savara Savara is JBoss project developed by Red Hat and employed in a commercial setting by a Cognizant business unit e Cognizant business unit. 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 OOI 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 summarise 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 formalised), 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, 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 Team [a]. 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.

7 Conclusion

This reports provides a comprehensive overview of the support of behavioral types in programming languages, taking into accout both static and runtime aspects. Overall, the report gives consistent evidence of the fact that behavioral types, despite their recent appearence, have sprinkled significant interest in the realm of programming languages and have the potential for being adopted in very different contexts. As shown in Section 6, behavioral types are also having an impact on the way software is designed, not just developed. It is to be noted, however, that in many cases the proposed solutions rely on modifications of type systems and therefore on some native support provided by language compilers and other developer tools. As these tools tend to evolve slowly, it is therefore interesting to envisage less invasive, yet concrete forms of integration of behavioral types into existing programming language. This can be achieved either by means of explicit code annotations in the form of comments or pragma directives or by means of syntax extensions to be handled by pre-processing.

Another issue is that the experimental integrations of behavioral types into programming languages have been mostly focused on "simple" types used for ensuring basic safety properties, such as the absence of communication errors. In this respect, the report from *WG1 – Foundations* shows that an increasing number of behavioral type theories rely on more informative types in order to enforce stronger properties. It is not evident that the integrations that have succeeded so far scale smoothly when richer types are considered. This calls for an effort to investigate the integration of such richer types into mainstream programming languages.

Bibliography

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*, OOPSLA'09, pages 1015–1022, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4. DOI 10.1145/1639950.1640073.

David S Allison, Miriam AM Capretz, Hany F EL 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.

Rajeev Alur and Doron Peled, editors. *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, 2004. Springer. ISBN 3-540-22342-8.

Nuno Alves, Raymond Hu, Nobuko Yoshida, and Pierre-Malo Deniélou. Secure execution of distributed session programs. In *Proceedings of PLACES'10*, volume 69 of *EPTCS*, pages 1–11, 2010. DOI 10.4204/EPTCS.69.1.

D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason. In *DALT X. Revised, Selected and Invited Papers*, volume 7784 of *LNAI*. Springer, 2012.

D. Ancona, M. Barbieri, and V. Mascardi. Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In *SAC 2013*. ACM, 2013a.

D. Ancona, V. Mascardi, and M. Barbieri. Global types for dynamic checking of protocol conformance in multi-agent systems. Technical report, DIBRIS, 2013b. Submitted for journal publication.

D. Ancona, V. Mascardi, and M. Barbieri. Global types for dynamic checking of protocol conformance of multi-agent systems. Technical report, University of Genova, DIBRIS, 2013c. Extended version of *D. Ancona, M. Barbieri, and V. Mascardi. Global Types for Dynamic Checking of Protocol Conformance of Multi-Agent Systems (Extended Abstract). In P. Massazza, editor, 13th Italian Conference on Theoretical Computer Science (ICTCS 2012), pages 39-43, 2012.*

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

F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.

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 *LNCS*, pages 659–680. Springer-Verlag, 2008a. DOI 10.1007/978-3-540-68679-8_41.

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR '08*, volume 5201 of *LNCS*, 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*, FirstView:1–57, 4 2013. ISSN 1469-8072. DOI 10.1017/S0960129512000886.

Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *FMOODS*, volume 7892 of *LNCS*, pages 50–65. Springer, 2013.

Viviana Bono and Luca Padovani. Typing Copyless Message Passing. *Logical Methods in Computer Science*, 8:1–50, 2012. ISSN 1860-5974. DOI 10.2168/LMCS-8(1:17)2012.

Viviana Bono, Chiara Messa, and Luca Padovani. Typing Copyless Message Passing. In *Proceedings of the 20th European Symposium on Programming (ESOP'11)*, volume LNCS 6602, pages 57–76. Springer, 2011. DOI 10.1007/978-3-642-19718-5_4.

Viviana Bono, Luca Padovani, and Andrea Tosatto. Polymorphic Types for Leak Detection in a Session-Oriented Functional Language. In *Proceedings of 2013 IFIP Joint International Conference on Formal Techniques for Distributed Systems*, volume LNCS 7892, pages 83–98. Springer, 2013. DOI 10.1007/978-3-642-38592-6_7.

R. H. Bordini, J. F. Hübner, and M. 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 Wirsing and Hölzl [2011], pages 282–301. ISBN 978-3-642-20400-5.

Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors. *Web Services Foundations*. Springer, 2014. ISBN 978-1-4614-7517-0, 978-1-4614-7518-7.

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

D. Brand and P. 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 Lumpe and Vanderperren [2007], pages 34–50. ISBN 978-3-540-77350-4.

Mario Bravetti and Gianluigi Zavattaro. Contract compliance and choreography conformance in the presence of message queues. In Bruni and Wolf [2009], pages 37–54. ISBN 978-3-642-01363-8.

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, 2012.

Roberto Bruni and Karsten Wolf, editors. *Web Services and Formal Methods, 5th International Workshop, WS-FM 2008, Milan, Italy, September 4-5, 2008, Revised Selected Papers*, volume 5387 of *Lecture Notes in Computer Science*, 2009. Springer. ISBN 978-3-642-01363-8.

Alexandre Caldeira and Vasco T. Vasconcelos. Bica. http://gloss.di.fc.ul.pt/bica/, 2011.

Joana Campos and Vasco T. Vasconcelos. Channels as objects in concurrent object-oriented programming. In Honda and Mycroft [2010], pages 12–28.

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. DOI 10.1016/j.tcs.2008.09.016.

Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Giacobazzi and Cousot [2013], pages 263–274. ISBN 978-1-4503-1832-7.

Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A theoretical basis of communication-centred concurrent programming. Published in CDL, 2006.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.

G. Castagna, M. Dezani-Ciancaglini, and L. 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.

CDLPRIMER. Web Services Choreography Description Language: Primer 1.0. `http://www.w3.org/TR/ws-cdl-10-primer/`, 2006.

B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007. ISSN 1094-3420. DOI `10.1177/1094342007078442`. URL `http://dx.doi.org/10.1177/1094342007078442`.

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 OOPSLA'05*, pages 519–538. ACM, 2005. ISBN 1-59593-031-0.

Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 7173 of *Lecture Notes in Computer Science*, pages 25–45. Springer, 2011a. DOI `http://dx.doi.org/10.1007/978-3-642-30065-3_2`.

Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, pages 25–45, 2011b.

Chor. Programming Language. `http://www.chor-lang.org/`.

E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In Marcello Bonsangue and Einar Broch Johnsen, editors, *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer, 2007. DOI `10.1007/978-3-540-72952-5_1`.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proceedings of PPDP'12*, pages 139–150. ACM, 2012. DOI `10.1145/2370776.2370794`.

Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012. ISBN 978-3-642-32939-5.

Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, LNCS. Springer, 2012.

Jolie development team. Jolie Programming Language. `http://www.jolie-lang.org/`.

Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. $|_{doos}$: a Distributed Object-Oriented language with Session types. In Rocco De Nicola and Davide Sangiorgi, editors, *TGC 2005*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005. ISBN 3-540-30007-4. DOI 10.1007/11580850_16.

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In Dave Thomas, editor, *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006. DOI 10.1007/11785477_20.

Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Elena Giachino, and Nobuko Yoshida. Bounded Session Types for Object-Oriented Languages. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO'06*, volume 4709 of *LNCS*, pages 207–245. Springer-Verlag, 2007. DOI 10.1007/978-3-540-74792-5_10.

Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and Session Types. *Information and Computation*, 207(5):595–641, 2009. DOI 10.1016/j.ic.2008.03.028.

Theo D'Hondt, editor. *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, 2010. Springer. ISBN 978-3-642-14106-5. DOI 10.1007/978-3-642-14107-2.

Sophia Drossopoulou, Mariangiola Dezani-Ciancaglini, and Mario Coppo. Amalgamating the Session Types and the Object Oriented Programming Paradigms. In *MPOOL'07*, 2007. URL http://homepages.fh-regensburg.de/~mpool/mpool07/programme.html.

Qualit e Cognizant business unit. Zero Deviation Life Cycle. http://0deviation.com/.

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 (PLDI), Berlin, Germany, June 17-19, 2002*, 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 EuroSys'06*, pages 177–190. ACM, 2006. DOI 10.1145/1217935.1217953.

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

Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Approved for standard, 2002.

Cédric Fournet, C. A. R. Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-free conformance. In Alur and Peled [2004], pages 242–254. ISBN 3-540-22342-8.

Carlo A. Furia and Sebastian Nanz, editors. *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, volume 7304 of *Lecture Notes in Computer Science*, 2012. Springer. ISBN 978-3-642-30560-3. DOI 10.1007/978-3-642-30561-0.

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. DOI 10.1017/S0956796809990268.

Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Proceedings of POPL'10*, pages 299–312. ACM Press, 2010. DOI 10.1145/1706299.1706335. Extended version: http://arxiv.org/abs/1205.5344.

Roberto Giacobazzi and Radhia Cousot, editors. *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, 2013. ACM. ISBN 978-1-4503-1832-7.

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

Kohei Honda and Alan Mycroft, editors. *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software*, volume 69 of *EPTCS*, 2010.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL '08*, pages 273–284. ACM, 2008.

Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In Natarajan and Ojo [2011], pages 55–75. ISBN 978-3-642-19055-1. DOI 10.1007/978-3-642-19056-8_4.

Kohei Honda, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Verification of mpi programs using session types. In Träff et al. [2012], pages 291–293. ISBN 978-3-642-33517-4.

Kohei Honda, , Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Deniélou, , and Nobuko Yoshida. Structuring communication with session types. In *COB'12*, Lecture Notes in Computer Science. Springer, to appear.

Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *Proceedings of ECOOP'08*, volume LNCS 5142, pages 516–541, 2008. DOI 10.1007/978-3-540-70592-5_22.

Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java. In D'Hondt [2010], pages 329–353. ISBN 978-3-642-14106-5. DOI 10.1007/978-3-642-14107-2_16.

Raymond Hu, Rumyana Neykova, Nobuko Yoshida, and Romain Demangeon. Practical Interruptible Conversations: Distributed Dynamic Verification with Session Types and Python. In *RV'13*, volume 8174 of *LNCS*, 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 Honda and Mycroft [2010], pages 74–91.

Ocean Observatories Initiative. Scribble OOI derivalables. https://confluence.oceanobservatories.org/display/CIDev/Identify+required+Scribble+extensions+for+advanced+scenarios+of+R3+COI.

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 (PPDP'12)*, pages 151–162. ACM, 2012. DOI 10.1145/2370776.2370796. URL http://www.di.unito.it/~padovani/Papers/JaksicPadovani12.pdf.

Svetlana Jakšić and Luca Padovani. Exception Handling for Copyless Messaging. *Science of Computer Programming*, ??(??):??–??, 2013. ISSN 0167-6423. DOI 10.1016/j.scico.2013.05.001.

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.

J. Kopecky, T. Vitvar, C. Bournez, and J. Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *Internet Computing, IEEE*, 11(6):60–67, Nov.-Dec. ISSN 1089-7801. DOI 10.1109/MIC.2007.134.

I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *SEFM*, pages 323–332. IEEE, 2008.

Cristina Videira Lopes and Kathleen Fisher, editors. *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, 2011a. ACM. ISBN 978-1-4503-0940-0.

Cristina Videira Lopes and Kathleen Fisher, editors. *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, 2011b. ACM. ISBN 978-1-4503-0942-4.

Markus Lumpe and Wim Vanderperren, editors. *Software Composition, 6th International Symposium, SC 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers*, volume 4829 of *Lecture Notes in Computer Science*, 2007. Springer. ISBN 978-3-540-77350-4.

Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco T. Vasconcelos, , and Nobuko Yoshida. Specification and verification of protocols for mpi programs. 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. PLACES 2013—6th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, 2013b. URL http://www.di.fc.ul.pt/~vv/papers/marques.martins.etal_deductive-verification-mpi-programs.pdf.

Viviana Mascardi and Davide Ancona. Attribute global types for dynamic checking of protocols in logic-based multiagent systems (technical communication). To appear in Theory and Practice of Logic Programming, On-line Supplement, as technical communication of the ICLP 2013 conference, 2013.

J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In *ATAL*, pages 347–360. Springer Verlag, 1995.

Jan Mendling and Michael Hafner. From inter-organizational workflows to process execution: Generating bpel from ws-cdl. In *OTM Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 506–515. Springer, 2005. DOI http://dx.doi.org/10.1007/11575863_70.

Wolfgang De Meuter and Gruia-Catalin Roman, editors. *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6721 of *Lecture Notes in Computer Science*, 2011. Springer. ISBN 978-3-642-21463-9.

Mira Mezini, editor. *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, 2011. Springer. ISBN 978-3-642-22654-0.

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, 2008. URL http://run.unl.pt/handle/10362/1792.

Filipe Militão and Luís Caires. An exception aware behavioral type system for object-oriented programs. In *Proceedings of INFORUM 2009*, 2009. URL http://www.cs.cmu.edu/~foliveir/papers/corta2009.pdf.

Fabrizio Montesi. Process-aware web programming with jolie. In Shin and Maldonado [2013], pages 761–763. ISBN 978-1-4503-1656-9.

Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2013. DOI http://dx.doi.org/10.1007/978-3-642-40184-8_30.

Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Composing services with jolie. In *ECOWS*, pages 13–22. IEEE Computer Society, 2007. DOI http://doi.ieeecomputersociety.org/10.1109/ECOWS.2007.7.

Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In Bouguettaya et al. [2014], pages 81–107. ISBN 978-1-4614-7517-0, 978-1-4614-7518-7.

Dimitris Mostrous. *Session Types in Concurrent Calculi: Higher-Order Processes and Objects*. PhD thesis, Imperial College London, November 2009.

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 316–332, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00589-3. DOI 10.1007/978-3-642-00590-9_23. URL http://dx.doi.org/10.1007/978-3-642-00590-9_23.

Raja Natarajan and Adegboyega K. Ojo, editors. *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings*, volume 6536 of *Lecture Notes in Computer Science*, 2011. Springer. ISBN 978-3-642-19055-1.

Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978. ISSN 0001-0782. DOI 10.1145/359657.359659. URL http://doi.acm.org/10.1145/359657.359659.

Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Proceedings of PADL'04*, volume LNCS 3057, pages 56–70. Springer, 2004.

Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: Local Verification of Global Protocols. In *RV'13*, volume 8174 of *LNCS*, pages 358–363. Springer, 2013.

Nicholas Ng, Nobuko Yoshida, Olivier Pernet, Raymond Hu, and Yiannos Kryftis. Safe parallel programming with session java. In Meuter and Roman [2011], pages 110–126. ISBN 978-3-642-21463-9. DOI 10.1007/978-3-642-21464-6_8.

Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session c: Safe parallel programming with message optimisation. In Furia and Nanz [2012], pages 202–218. ISBN 978-3-642-30560-3. DOI 10.1007/978-3-642-30561-0_15.

Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *TOOLS*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012b.

Nicholas Ng, Nobuko Yoshida, Xinyu Niu, and Kuen Hung Tsoi. Session types: towards safe and fast reconfigurable programming. *SIGARCH Computer Architecture News*, 40(5):22–27, 2012c.

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

OOI. The Ocean Observatories Initiative. `http://oceanobservatories.org/`.

Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 25–36, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7. DOI `10.1145/1411286.1411290`.

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*, WWW '07, pages 973–982, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. DOI `10.1145/1242572.1242704`. URL `http://doi.acm.org/10.1145/1242572.1242704`.

A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *MAAMAW'96*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996.

Red Hat JBoss. JBoss Community Scribble homepage. `http://www.jboss.org/scribble`.

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 First Workshop on Advances in Message Passing*, 2010.

JBoss Savara. JBoss Savara Project homepage. `http://www.jboss.org/savara`.

Aamir Shafi, Bryan Carpenter, and Mark Baker. Nested parallelism for multi-core hpc systems using java. *J. Parallel Distrib. Comput.*, 69(6):532–545, 2009. ISSN 0743-7315. DOI `10.1016/j.jpdc.2009.02.006`. URL `http://dx.doi.org/10.1016/j.jpdc.2009.02.006`.

Sung Y. Shin and José Carlos Maldonado, editors. *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, 2013. ACM. ISBN 978-1-4503-1656-9.

Guy L. Steele. Parallel programming and parallel abstractions in fortress. In *Proceedings of the 8th international conference on Functional and Logic Programming*, FLOPS'06, pages 1–1, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33438-6, 978-3-540-33438-5. DOI `10.1007/11737414_1`. URL `http://dx.doi.org/10.1007/11737414_1`.

Zachary Stengel and Tevfik Bultan. Analyzing Singularity Channel Contracts. In *Proceedings of ISSTA'09*, pages 13–24. ACM, 2009. ISBN 978-1-60558-338-9. DOI `10.1145/1572272.1572275`.

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

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in Plaid. In Lopes and Fisher [2011a], pages 713–732. ISBN 978-1-4503-0940-0. DOI `10.1145/2048066.2048122`.

Joshua Sunshine, Sven Stork, Karl Naden, and Jonathan Aldrich. Changing state in the Plaid language. In Lopes and Fisher [2011b], pages 37–38. ISBN 978-1-4503-0942-4. DOI `10.1145/2048147.2048166`.

Scribble Team. Scribble Project github homepage, a. `http://www.scribble.org`.

Scribble Team. Scribble Language Reference, b. `https://github.com/scribble/scribble-spec`.

Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors. *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, volume 7490 of *Lecture Notes in Computer Science*, 2012. Springer. ISBN 978-3-642-33517-4.

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. DOI 10.1016/j.tcs.2006.06.028.

Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving Copyless Message Passing. In *Proceedings of APLAS'09*, LNCS 5904, pages 194–209. Springer, 2009. DOI 10.1007/978-3-642-10672-9_15.

Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking Heaps That Hop with Heap-Hop. In *Proceedings of TACAS'10*, LNCS 6015, pages 275–279. Springer, 2010. DOI 10.1007/978-3-642-12002-2_23.

Martin Wirsing and Matthias M. Hölzl, editors. *Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, volume 6582 of *Lecture Notes in Computer Science*. Springer, 2011. ISBN 978-3-642-20400-5.

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In Mezini [2011], pages 459–483. ISBN 978-3-642-22654-0. DOI 10.1007/978-3-642-22655-7_22.

Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007. URL http://dx.doi.org/10.1016/j.entcs.2007.02.056.