Fundamenta Informaticae XX (2005) 1–16
IOS Press

Typing the Behavior of Software Components using Session Types

Antonio Vallecillo
Dept. of Computer Science, University of Málaga

Vasco T. Vasconcelos
Dept. of Informatics, Faculty of Sciences, University of Lisbon

António Ravara
CLC and Dept. of Mathematics, IST, Technical University of Lisbon

Abstract. This paper proposes the use of session types to extend with behavioural information the simple descriptions usually provided by software component interfaces. We show how session types allow not only high level specifications of complex interactions, but also the definition of powerful interoperability tests at the protocol level, namely compatibility and substitutability of components. We present a decidable proof system to verify these notions, which makes our approach of a pragmatic nature.

1. Introduction

Component-Based Software Development (CBSD) is gaining recognition as the key technology for the construction of high-quality, evolvable, large software systems, developed in timely and affordable manners. CBSD advocates the development and usage of plug-and-play reusable software, with the goal of reducing developing costs and efforts, while improving the flexibility and reliability of the final application due to the (re)use of software components already tested and validated.

In CBSD, components are prefabricated pieces, perhaps developed at different times, by different teams, and possibly with different uses in mind. The development effort now becomes one of gradual discovery about the components, their capabilities, their internal assumptions, and the incompatibilities that arise when they are used in concert. Therefore, the notions of substitutability and compatibility of software components play a critical role in CBSD, since we need to be able to check whether a given

Address for corespondance: Antonio Vallecillo <av@lcc.uma.es>

component can successfully replace another in a particular application, or whether the behaviour of two components is compatible for them to inter-operate.

In general, components are described by means of their *interfaces*, which define their functionality and capabilities independently from any particular implementation. Component interfaces currently provide this information in terms of the signature of the services offered by the component, and commercial component platforms (such as CORBA, EJB, or .NET) provide the basic infrastructure for component interoperability based on them. This allows to sort out most of the "plumbing" issues when putting components together to build applications. However, all parties are starting to recognise that signature interoperability is not sufficient for ensuring the correct development of component-based applications in open systems [23].

Traditional approaches to overcome this limitation try to add *semantic* information to interfaces, using different notations (pre/post conditions, temporal logic, Petri nets, refinement calculus, etc.), and are also concerned about compatibility and substitutability of components (see [15] for a comprehensive survey). However, these proposals share a common drawback: the definitions of interoperability tests and of other behavioural properties of components and applications, based on their full semantic descriptions, are either undecidable or have a high computational complexity, what hinders their practical utility.

Half-way betwen signature and semantics approaches, lies another possibility that concentrates on the components' interactions with other components, defining their service access protocols, and the way they use other components' services. More than signature information, this approach allows the definition of compatibility and substitutability checks among components at a computational cost lower than other semantic tests.

Some authors have dealt with component interoperability at this level [5, 14, 22]—usually called the *protocol* level—, and have shown its benefits. However, the existing approaches, when decidable, still present some limitations:

- First, the description of the components' observable behaviour is not modular: each component is assigned a single protocol description, which defines all its interactions with the rest of the components in the system. This mixes up all interactions, and usually forces the introduction of irrelevant details into the protocol specification, e.g. the interleaving among unrelated interactions.
- Second, the computational complexity of most of the tests is high, due in part to the fact of having to check full protocols. Typical (pairwise) component interactions are very simple, and this should reflect in simpler compatibility tests.

In this paper we use the concept of *session types* [11–13, 19] for describing the dynamic behaviour of components. Sessions are partial protocol specifications, in which we only pay attention to the behaviour of a component's interface. Such an approach allows modular specification of the behaviour of the components, providing more than just signature information, and permits precise definitions of compatibility and substitutability tests. The main strength of our approach is that these notions are decidable and algorithmically checkable, at a low computational cost.

Furthermore, session types are *types*, and therefore supported by a type discipline. This is a key element of the structuring method that provides typability checks between a program implementing a protocol and a session type describing its intended use. Moreover, the framework also permits checking component substitutability based on the concept of session subtyping, which is decidable and computationally tractable, in contrast to the—when decidable—exponential tests that result from the use of traces or process algebras in protocol descriptions.

```
protocol Auctioneer {
  session withASeller =
    +{ selling: ![string, float];
       &{ sold: ?(float); end
        | notSold; end
       }
    }
  session withABidder =
    +{ register: &{ wannaBid: ?(string, float); ![boolean]; Bidding} }
  Bidding =
    &{ wannaBid: ?(string, float); ![boolean]; Bidding
     | itemSold: ?(string); Unregistering
     | youGotIt: ?(string, float); Unregistering
    }
  Unregistering =
    +{ unregister: end }
}
```

Figure 1. Distributed auction bidding.

Our work builds on that by Honda *et al.* [12, 13, 19], which initially introduced session types for describing structured communication. Protocol compatibility and substitutability tests are defined using the subtyping relation defined by Gay and Hole for session types [11]. In this paper we first complement those works by introducing the notion of *compatibility* between session types, prove some of its properties, and then study how session types can be successfully applied not only at the theoretical level, but also in a commercial environment such as the one that CORBA provides. In addition, we also discuss some implementation details, namely how to check that a given object implementation conforms to a session type that supposedly describes its behaviour.

The structure of this paper is as follows. After this introduction, Section 2 introduces the language we propose for describing component interactions, using an example application that will be used throughout the paper. Section 3 introduces the type discipline supporting the language, including an algorithmic subtyping relation for session types. This relation also serve us to define the notion of compatibility between components. The application of our theoretical results to the particular case of CORBA is presented in Section 4. In Section 5 we discuss the sorts of tests that can be carried out with our proposal, both statically and during run-time. Finally, Section 6 relates our work to other similar approaches, and Section 7 points to future work.

2. Expressing component interactions via protocols

We illustrate the usage of the language via an example. Consider a distributed auction system, with three kinds of players: *sellers* that want to sell items, an *auctioneer* that sells items on their behalf, and *bidders* that bid for an item being auctioned.

The protocol that describes the interactions of the auctioneer with a seller (Figure 1) is simple: there is only one operation that sellers may invoke on an auctioneer—selling—where they provide the bidder with a description of the item to be sold (a **string**), and the minimum price they are willing to sell the item for (a **float**). This accounts for the +{ selling: ![**string**, **float**]; part of the protocol. Sellers then wait on the outcome of their request. Two things can happen: either the item was sold (in which case the seller gets the price the item was sold for), or the item was not sold. The first case is modeled by the acceptance of operation sold; the second by the operation notSold. In either case the protocols halts, as indicated by the **end** mark.

We describe protocols *centered on the clients*, sellers and bidders in this case. The distinction between the outbound operation +{selling: ... }, and the inbound operation &{sold: ... | notSold: ... } must be stressed: the former denotes an operation invoked by any seller (and thus provided by the auctioneer), the latter describes an operation provided by a seller (and thus invoked by an auctioneer).

The auctioneer's protocol with a bidder is slightly more complex. Bidders start by registering themselves at the auctioneer, then enter an interactive bidding session, and eventually unregister, thus leaving the protocol. Register is an operation without arguments. Upon registering, the bidder gets a bidding proposal (wannaBid) containing a description of an item (a **string**) and a price (a **float**); to which they answer "I am interested" or "I skip" (a **boolean**). This accounts for the &{ wannaBid: ?(**string**, **float**); ![**boolean**]; part of the protocol. The interactive session starts then, as described by the Bidding "loop": the bidder must be ready for three different kinds of requests coming from the auctioneer: new wannaBid challenges (either for the same item or for a distinct one), and two different acknowledgments. The auctioneer request itemSold says that the given item is no longer for sale (it may have been sold, or the the price may have got below the minimum required by the seller); request youGotIt comes with the item description and the final price. Notice that wannaBid operations brings the protocol back to the Bidding loop, whereas the two acknowledgments take the protocol to the unregistering phase and then to halt.

Consider now the protocol for a potential seller: it interacts with an auctioneer willing to accept the description of the item to be sold (a **string**), and its minimum price (a **float**). Then it waits for the client to provide the outcome: sold or not notSold. Below is a possible description.

```
protocol Seller {
  session withAnAuctioneer =
    &{ selling: ?(string, float);
      +{ sold: ![float]; end
       | notSold: end
      }
}
```

There is a close relationship between session Auctioneer::withASeller and session Seller::withAnAuctioneer: where one says select (+) the other says branch (&), where one says output (!) the other says input (?). In fact, the two sessions are *complementary* or *dual* (the exact definition is in Section 3). Duality is what guarantees that *sessions do not go wrong*: it precludes the standard "message not understood" error (operation not provided, wrong number of arguments, or wrong sort for an argument), and also problems derived from misunderstandings of the next operation in a protocol (both partners output at a given point; one partner **end**s the protocol whereas the other requests a different operation).

As a last example consider a more apt seller, that after initiating the selling process, is able to process three kinds of requests: the familiar sold/notSold, as well as the new lowerYourPrice, to which the seller

$$
\begin{array}{rcl}
\text{Protocol} & ::= & \textbf{protocol}\ X\ \{\text{Sessions}\} \\
\text{Sessions} & ::= & \textbf{session}\ X = T \ldots\ \textbf{session}\ X = T \\
T & ::= & \&\{m_1 : T_1 \mid \cdots \mid m_n : T_n\} \ \mid\ +\{m_1 : T_1 \mid \cdots \mid m_n : T_n\} \ \mid \\
 & & ?(\tilde{T}); T \ \mid\ ![\tilde{T}]; T \ \mid\ ?(\widetilde{\text{sort}}); T \ \mid\ ![\widetilde{\text{sort}}]; T \ \mid\ X \ \mid\ \mu X.T \ \mid\ \textbf{end} \\
\text{sort} & ::= & \textbf{string} \ \mid\ \textbf{float} \ \mid\ \textbf{boolean}
\end{array}
$$

Figure 2. A grammar for describing protocols.

may refuse or assent. In the latter case the seller sends a new price, and the selling process restarts. Here is a possible definition.

```
protocol SuperSeller {
  session withAnAuctioneer =
    &{ selling: ?(string, float); Selling }
    Selling =
      +{ sold: ![float]; end
       | notSold: end
       | lowerYourPrice:
           &{ ok: ?(float); Selling
            | noWay: end}
      }
}
```

Can a SuperSeller try to sell an item to an Auctioneer? The SuperSeller has more behaviour than the Seller: he can conduct all the sessions a Seller conducts (basically, selling-sold and selling-notSold), but also more sophisticated sessions (such as selling-lowerYourPrice-ok-lowerYourPrice-ok-sold). Essentially, SuperSeller::withAnAuctioneer has more-or-equal selections (+), and less-or-equal branchings (&); session SuperSeller::withAnAuctioneer is a *subtype* of session Seller::withAnAuctioneer: we write SuperSeller::withAnAuctioneer $\leq$ Seller::withAnAuctioneer (the exact definition is in Section 3).

What makes *session* SuperSeller::withAnAuctioneer *compatible* with session Auctioneer::withASeller? The fact that the former is a subtype of a type (Seller::withAnAuctioneer) that is dual to the latter. In this case we write SuperSeller::withAnAuctioneer $\bowtie$ Auctioneer::withASeller. Finally we may say that *protocol* SuperSeller is *compatible* with protocol Auctioneer since there is a session in the former (withAnAuctioneer) that is compatible with a session in the latter (withASeller). SuperSeller is compatible with Auctioneer, only that part of its programmed behaviour never gets excited by the basic Auctioneer.

3. A type discipline for sessions

Two are the key concepts we are interested in this paper: *component substitutability* and *component compatibility*. The former refers to the ability of a component to replace another in such way that the change goes unnoticeable by the clients. The latter defines when two components can work properly together,

$$\frac{T \leq S \in \Sigma}{\Sigma \vdash T \leq S} \quad \text{(S-ASSUMP)}$$

$$\Sigma \vdash \textbf{end} \leq \textbf{end} \quad \text{(S-END)}$$

$$\frac{\Sigma \vdash T \leq S}{\Sigma \vdash ?(\widetilde{\text{sort}}); T \leq ?(\widetilde{\text{sort}}); S} \quad \text{(S-SORTIN)}$$

$$\frac{\Sigma \vdash T \leq S}{\Sigma \vdash ![\widetilde{\text{sort}}]; T \leq ![\widetilde{\text{sort}}]; S} \quad \text{(S-SORTOUT)}$$

$$\frac{\Sigma \vdash T \leq S \qquad \Sigma \vdash T_i \leq S_i \qquad \forall i \in \{1, \ldots, n\}}{\Sigma \vdash ?(T_1 \ldots T_n); T \leq ?(S_1 \ldots S_n); S} \quad \text{(S-TYPEIN)}$$

$$\frac{\Sigma \vdash T \leq S \qquad \Sigma \vdash S_i \leq T_i \qquad \forall i \in \{1, \ldots, n\}}{\Sigma \vdash ![T_1 \ldots T_n]; T \leq ![S_1 \ldots S_n]; S} \quad \text{(S-TYPEOUT)}$$

$$\frac{\Sigma \vdash T_i \leq S_i \qquad \forall i \in \{1, \ldots, n\}}{\Sigma \vdash \&\{m_1 : T_1 \mid \cdots \mid m_n : T_n\} \leq \&\{m_1 : S_1 \mid \cdots \mid m_m : S_{n+k}\}} \quad \text{(S-BRANCH)}$$

$$\frac{\Sigma \vdash T_i \leq S_i \qquad \forall i \in \{1, \ldots, n\}}{\Sigma \vdash +\{m_1 : T_1 \mid \cdots \mid m_m : T_{n+k}\} \leq +\{m_1 : S_1 \mid \cdots \mid m_n : S_n\}} \quad \text{(S-SELECT)}$$

$$\frac{\Sigma, \mu X.T \leq S \vdash \mathsf{unwind}(\mu X.T) \leq S}{\Sigma \vdash \mu X.T \leq S} \quad \text{(S-RECL)}$$

$$\frac{\Sigma, S \leq \mu X.T \vdash S \leq \mathsf{unwind}(\mu X.T)}{\Sigma \vdash S \leq \mu X.T} \quad \text{(S-RECR)}$$

Figure 3. Subtyping system.

if connected. These concepts can be considered as the two flip sides of the component interoperability coin. In this section we precisely define these concepts within the framework of a type discipline.

The language of types The language we use to describe protocols is generated by the grammar in Figure 2, where $\tilde{P}$ describes a sequence of zero or more P. The intended meaning of most constructors is exemplified in the previous section. The recursive type constructor $\mu X.T$ constitutes the only binder in the language, binding variable X in type T. Substitution $T[S/X]$, of a type S for a variable X in a type T, is defined accordingly, and so is alpha-equivalence. We work up to alpha-equivalence.

The examples in the previous section are defined with equations, rather than with explicit recursion. Here we adopt explicit recursion, since it simplifies the theory. For example, session SuperSeller:: withAnAuctioneer is translated into

&{ selling: ?(**string**, **float**); T }
where T is μX. + { sold: ![**float**]; **end** | notSold: **end** | lowerYourPrice: T' }
and T' is &{ ok: ?(**float**); X | noWay: **end** }.

$$
\begin{aligned}
\overline{?(\tilde{S});T} &= !{[\tilde{S}]};\overline{T} \\
\overline{![\tilde{S}];T} &= ?(\tilde{S});\overline{T} \\
\overline{\&\{m_1:T_1 \mid \cdots \mid m_n:T_n\}} &= +\{m_1:\overline{T}_1 \mid \cdots \mid m_n:\overline{T}_n\} \\
\overline{+\{m_1:T_1 \mid \cdots \mid m_n:T_n\}} &= \&\{m_1:\overline{T}_1 \mid \cdots \mid m_n:\overline{T}_n\} \\
\overline{\mathbf{end}} &= \mathbf{end} \\
\overline{X} &= X \\
\overline{\mu X.T} &= \mu X.\overline{T}
\end{aligned}
$$

Figure 4. The dual of a type.

Subtyping Figure 3 defines a subtyping relation for session types, as in [10]. As usual, T is a subtype of S, written $T \leq S$, if T can be used in any context where S is used. Therefore, T should have *more-or-equal selections* ($+$) and *less-or-equal branchings* ($\&$). Branching and selection behave in a co-variant manner with respect to subtyping, which is also co-variant for the input, but (as usual) contra-variant for the output.

The rules for algorithmic subtyping are of the form $\Sigma \vdash T \leq S$, where Σ is finite set of inequalities $T \leq S$, meaning that type T is a subtype of type S, assuming the inequalities in Σ, and where $\mathsf{unwind}(\mu X.T) \stackrel{\text{def}}{=} T[\mu X.T/X]$. When $\emptyset \vdash T \leq S$ is derivable, we simply write $T \leq S$.

Using this operator, in our context, an expression '$T \leq S$' will (indistinctly) mean: "*T is more restrictive than S*"; "*S is more general than T*"; "*T can (safely) replace S*"; or "*S is substitutable by T*".

Type duality Section 2 hints that sessions Auctioneer::withASeller and Seller:: withAnAuctioneer are *dual* because where one says select (+) the other says branch (&), where one says output (!) the other says input (?). To obtain the dual type $\overline{T}$ of a type T, we use the inductive definition in Figure 4, taken from [13], where S stands for a sort or for a type. To check that two given types S and S' are dual to each other, we obtain the dual $\overline{S}$ of S, and check that both $\overline{S} \leq S'$ and $S' \leq \overline{S}$.

Substitutability and Compatibility Based on the subtyping relation, we are finally in a position to define the concepts of substitutability and compatibility between session types. Substitutability is simply subtyping; T is compatible with S is T is a subtype of the dual of S.

Definition 3.1. Let T and S be session types. We say that:

1. *T can safely substitute S*, if $T \leq S$;
2. *T is compatible with S*, and write $T \bowtie S$, if $T \leq \overline{S}$.

Example: SupperSeller is compatible with Auctioneer According to the discussion in Section 2, we show that session $T \stackrel{\text{def}}{=}$ SuperSeller::withAnAuctioneer can safely substitute session $S \stackrel{\text{def}}{=}$ Seller::withAnAuctioneer, and that session T is compatible with session $U \stackrel{\text{def}}{=}$ Auctioneer::withASeller.

For the subtype part, recall that S is

$$\&\{\ \text{selling: } ?(\textbf{string}, \textbf{float}); S'\ \} \quad \text{where } S' \text{ is } +\{\ \text{sold: } ![\textbf{float}]; \textbf{end} \mid \text{notSold: } \textbf{end}\ \}$$

and that T is

$$\&\{\ \text{selling: } ?(\textbf{string}, \textbf{float}); T'\ \} \quad \text{where } T' \text{ is } \mu X.+\{\ \text{sold: } ![\textbf{float}]; \textbf{end} \mid \text{notSold: } \textbf{end} \mid \text{lowerYourPrice: } T''\ \} \quad \text{and } T'' \text{ is } +\{\ \text{ok: } ![\textbf{float}]; X \mid \text{noWay: } \textbf{end}\ \}.$$

To show that $T \leq S$, apply first the rule S-BRANCH, followed by S-SORTIN. We are left to prove that $T' \leq S'$. Applying rule S-RECL, yields $\mathsf{unwind}(T') \leq S'$. Finally, applying S-SELECT we are left with ![**float**]; **end**$\leq$![**float**]; **end** (which follows by S-SORTOUT; S-END), and **end**$\leq$ **end** (which follows by S-END). Notice how the recursive structure of SupperSeller was accounted for by rule S-RECL, and how its extra branch (lowerYourPrice) was ignored by rule S-SELECT.

For duality, recall that U is

$$+\{\ \text{selling: } ![\textbf{string}, \textbf{float}]; U'\ \} \quad \text{where } U' \text{ is } \&\{\ \text{sold: } ?(\textbf{float}); \textbf{end} \mid \text{notSold: } \textbf{end}\ \}.$$

To show that $T \bowtie U$ we compute $\overline{U}$, using the rules in Figure 4, to obtain S, and we are done, since we have already proved that $T \leq S$.

Results The main contribution of this paper is the decidability of the two notions we propose, component substitutability and component compatibility. Before that, we show a few other important properties of these notions.

Proposition 3.1. Subtyping $\leq$ is a preorder.

Proof:
Reflexivity follows by induction on the structure of the types involved, taking $k = 0$ in both S-BRANCH and S-SELECT rules. Gay and Hole ([11], Lemma 5) show that subtyping is transitive. □

It follows directly from the definition that duality is symmetric. Furthermore, it has an interesting relationship with subtyping.

Proposition 3.2. $T \leq \overline{S}$ if and only if $S \leq \overline{T}$.

Proof:
A straightforward induction on the structure of the types. □

We show that compatibility is also a symmetric relation; it is also preserved by subtyping. Notice that compatibility is neither reflexive nor transitive. It is not reflexive because duality is obviously not reflexive. It is not transitive because a session is never compatible with itself.

Proposition 3.3.

1. Compatibility $\bowtie$ is symmetric.
2. If $T \bowtie S$ and $U \leq T$, then $U \bowtie S$.

Proof:
The first clause is a direct consequence of lemma 3.2. The second clause is a consequence of the transitivity of subtyping, lemma 3.1. □

Finally, the decidability of the notions of substitutability and of compatibility of components result from a suitable reading of the rules in Figure 3.

Theorem 3.1. Subtyping and compatibility are decidable.

Proof:
For subtyping, read the rules in Figure 3 upwards, with the additional constraint that S-ASSUMP must be used if applicable, and that S-RECL must be used in preference to S-RECR. To check that $T \leq S$, apply the algorithm to $\emptyset \vdash T \leq S$. Termination of the algorithm is ensured by a suitable metric. See [10] for details.

To check the compatibility of types T and S, we build the dual $\overline{S}$ of S, using the definition in Figure 4, and then check the subtype relation $T \leq \overline{S}$. □

4. A case study

In this section we study how session types can be successfully applied not only at the theoretical level, but also in a commercial environment such as the one that CORBA provides.

CORBA is one of the major distributed object platforms. Proposed by the OMG (`http://www.omg.org`), the Object Management Architecture (OMA) attempts to define, at a high level of description, the various facilities required for distributed object-oriented computing. The core of the OMA is the Object Request Broker (ORB), a mechanism that provides transparency of object location, activation and communication. The Common Object Request Broker Architecture (CORBA) specification describes the interfaces and services that must be provided by compliant ORBs [18].

In the OMA model, objects provide services, and clients issue requests for those services to be performed on their behalf. The purpose of the ORB is to deliver requests to objects and return any output values back to clients, in a transparent way to the client and the server. Clients need to know the *object reference* of the server object. ORBs use object references to identify and locate objects to redirect requests to them. As long as the referenced object exists, the ORB allows the holder of an object reference to request services from it.

Even though an object reference identifies a particular object, it does not necessarily describe anything about the object's interface. Before an application can make use of an object, it must know what services the object provides. CORBA defines an Interface Description Language (IDL) to describe object interfaces, a textual language with a syntax resembling that of C++. The CORBA IDL provides basic data types (such as **short**, **long**, **float**), constructed types (**struct**, **union**) and template types (**sequence**, **string**). These are used to describe the interface of objects, defined by types, attributes and signatures

(parameters, return types and exceptions raised) of the object methods, all grouped into **interface** definitions. As an example, the Auctionner component presented in Section 2, is described in CORBA by the following interface.

```
interface Auctioneer {
  void selling (in string itemDesc, in float minPrice);
  string register (in Bidder b);    // returns an id for the bidder
  void unregister (in string id);
}
```

As we can see, the CORBA IDL allows us to describe the signature of the operations implemented by a component, but it falls short in several respects: a) it does not mention whom the operations are targeted at (selling is for sellers; register and unregister are for bidders); b) it does not describe the order in which operations must be invoked (register first; unregister then); c) it does not allow specifying the external operations required by the component (sellers must provide sold and notSold operations).

The rest of this section concentrates on how to add protocol information to the description of the CORBA object interfaces, using the language defined in Section 2.

CORBA object *protocols* are defined by two collections of interfaces and a collection of session types. The first collection describes the CORBA interfaces *provided* (i.e. implemented) by the component, each one under a **provides** heading. Second, we have the collection of external interfaces that the component *requires* from other objects when implementing its supported services, expressed by **uses** headings (there may be none in case the component does not require any external services). Finally, we find the specification of each role the component plays in its interactions with other components, expressed in terms of a collection of session types, each of them indicated by a **session** clause. The first two collections contain information at the signature level only, while the last one is in charge of specifying the dynamic aspects of the behaviour of the component. The grammar for describing *CORBA protocols* is obtained from that in Figure 2 by replacing the first production by the one below.[1]

$$\text{Protocol} \quad ::= \quad \textbf{protocol}\ X\ \{(\textbf{provides}\ X)^+\ (\textbf{uses}\ X)^*\ \text{Sessions}\}$$

The modelling technique that we propose for describing CORBA object interactions is the following.

1. In the CORBA IDL, methods have a return value and three kind of arguments: **in**, **out** and **inout**. In a method invocation from a client all **in** and **inout** arguments are sent in the same order they were declared in the IDL (which is reflected in the server's input action ?(...). In the method response (i.e. the ![...] output action from the server) the first sort is the sort of the return value, followed by the sorts of the **inout** and **out** arguments, in the same order they were declared. Likewise for method acceptance and reply.

2. Methods with no arguments are considered as if having one argument of sort **void**, to be added to those in Figure 2.

3. Special label **quit** is used in reactive servers to indicate the moments in which a client may disconnect from the session.

4. The client's invocation of method "s m(s_1,...,s_k)" is modelled as "m:?(s_1,...,s_k);![s];" inside a *branch* (&{...}) structure in the server side of the protocol.

[1] P^* describes a sequence of zero or more Ps; P^+ is PP^*.

```
protocol Bidder {
  provides Bidder
  uses Auctioneer
  session withAnAuctioneer =
    &{ register: ?(Bidder); ![string];
       +{ wannaBid: ![string, float]; ?(boolean); Bidding}
    }
  Bidding =
    +{ wannaBid: ![string, float]; ?(boolean); Bidding
     | itemSold: ![string]; ?(void); Unregistering
     | youGotIt: ![string, float]; ?(void); Unregistering
    }
  Unregistering =
    &{ unregister: ?(string); ![void]; end}
}
```

Figure 5. The CORBA Bidder protocol.

5. Analogously, the server's invocation of method "s m(s_1,...,s_k)" is modelled by "m:![s_1,...,s_k];?(s);" inside a *select* (+{...}) structure in the server protocol.

6. In case of methods that may raise exceptions, the return mechanism is different. Normal termination is modelled by a special label **success**, followed by an output action with the sort of the return argument. Exception raising is modelled by selecting a label with the name of the exception, followed by the output of the sort of the parameters that the exception returns. If a method may raise several exceptions, one label is used for each of them.

With this techniques, the protocol that defines the dynamic behaviour of object Bidder is shown in Figure 5, where Bidder components make use of the services provided by an Auctioneer component, whose interface is shown above.

It is important to note that session types describing CORBA interactions follow a reduced set of communication patterns, which are given by a sub-language on that in Figure 2. This is due to the fact that CORBA imposes restrictions on the communication patterns used by its objects, since client-server method invocation is the only mechanism allowed. Thus, a server can only offer its methods within a branch structure, then accept input parameters, output the results, and become either a client or a server again. But it can never start alternating inputs and outputs in an arbitrary manner.

Another issue worth noticing is the need to accommodate the CORBA particulars when describing protocols with session types. For instance, comparing the protocols in Figures 1 and 5 we see that CORBA methods return a value (the ?(**void**) not present in Figure 1), and that all arguments appearing in the CORBA IDL interfaces must be present in the protocol description (the Bidder parameter of method register, and the return of the string identifying the bidder, to be used with unregister). Although not needed in generic **protocol**s, they are necessary when describing the behaviour of the CORBA objects implementing a particular CORBA interface.

5. Checking protocols

As mentioned in the introduction, we are specially interested in checking component substitutability and compatibility, based on the information now available. We distinguish between static and dynamic checks. The first ones are carried out during the design time of the applications and are based on the description of their constituent components. Dynamic checks are needed when there is no behavioral information available, as it happens when dealing with back-box component instances—whose internals are not accessible—but we still want to check that its behavior conforms to a given protocol.

Statically checking a component against a session type amounts to type check the program code describing the component: Honda, Vasconcelos, and Kubo [13] describe a type system for a π-calculus based language; Vasconcelos, Ravara, and Gay [20] present a type checking algorithm for a functional multithreaded language.

There are many situations in which protocol compatibility has to be checked at run time, as it happens for instance when we are programming in a language for which we do not know how to type check conformance, or when we do not have access to the source code (this is the common case in component-based development environments, specially if we are using off-the-shelf binary components, whose internals cannot be accessed). In these cases, protocol compatibility can be checked by intercepting exchanged messages and verifying their correctness with regard to the current state of the components. This sort of information can be used in order to prevent illegal or incompatible messages to reach destination components, avoiding incompatibility issues. In this way, system inconsistency situations can be detected before they happen, and the appropriate exceptions or errors can be raised.

Such a mechanism can be implemented in CORBA using a reflective facility that some ORB vendors provide: interceptors [18] (also called *filters*) that allow the interception and observation of the messages exchanged among components. This mechanism allows a programmer to specify additional code to be executed before or after the normal execution of an operation. Such code may perform security checks, provide debugging traps or information, maintain an audit trail, etc. In our case, for each CORBA object, a filter can be defined that captures incoming and outgoing messages, reproduces its run-time trace, and checks that received messages are compatible with the behavior defined for that object. Basically, the interceptor for a particular session type builds an automata whose arcs are labeled with the constructors of types T in Figure 2 and checks that every incoming or outgoing message is valid with regard to this automata. In case a violation of the protocol is found, the invalid message is returned to the originator, using the interceptors mechanisms. Reference [6] provides further information on this sort of tools.

The schema described above can also be used to check *conformance to specifications*, that is, check that an implementation of a component conforms to a given specification of its intended behavior. This is specially important when we are dropping a new component in a given system, and we want to check that it will not violate the protocols of the components it communicates with. Even if programming language used to code the new component provides for static session type-checking, we may still need run-time checks, for the "rest of the system" may be composed of black-box components, whose code is inaccessibly.

6. Related work

As mentioned in the introduction, several authors have provided a number of proposals that try to overcome the limitations that current IDLs present, defining extensions that usually cope with the semantic aspects of component interfaces and behaviour. We do not cite here the proposals that try to deal with the full operational semantics of components (the interested reader can consult [15]), just the ones that cover the specification of the components' service access protocols.

These proposals use different notations for specifying protocols, from finite state machines to process algebras (see, e.g., [3, 5, 14, 17, 21, 22]). However, they all share some limitations. First, they do not allow the modular description of the protocols. Second, the compatibility and substitutability tests that they provide either are not decidable or do not have a tractable computational complexity. Finally, none of them are directly supported by a type discipline. Our proposal helps solve these problems, at the cost of sacrificing some expressiveness—just pairwise component interactions can be expressed in terms of session types (see Section 7).

Some representative examples of approaches that use different notations for representing the observable behaviour of components follow. In the first place, Doug Lea proposed PSL [14], an extension of the CORBA IDL to describe the protocols associated to an object's methods. This approach is based on logical and temporal rules relating situations, each of which describes potential states with respect to the roles of components, attributes, and events. Although it is an expressive approach, it does not account for the services an object may need from other objects, neither it is supported by standard proving tools.

Protocol Specifications [22] is a more general approach for specifying component service protocols that describe both the services offered and required by components. It is based on using finite state machines and allows components to be easily checked for protocol compatibility. The simplicity that allows the easy checking also makes it too rigid and lacking expressiveness for general usage in more complex open and distributed environments.

Bastide, and Sy [3] use Petri nets to describe the behaviour of CORBA objects, providing their full operational semantics, and supported by proving tools. This is a very powerful and expressive approach, that has been successfully used to detect inconsistencies in some CORBA services commercial implementations [2]. Bastide's proposal allows much richer information to be included in the objects' behaviourally descriptions, which may be required in some cases. However, the main problems of this proposal are the lack of modularity in the description of the protocols, and the decidability and computational complexity of the tests.

Message Sequence Charts (MSC) is also a notation that permits the description of the interactions among components, and is now part of UML 2.0. MSCs are very expressive for describing protocol interactions, but they do not allow to prove properties of the system. An interesting line of research would be to "type" MSCs using session types, hence allowing substitutability and compatibility tests between components whose observable behaviour is described with MSCs.

Braciali, Brogi, and Turini [4] also sacrifice expressiveness in order to achieve modularity and computational tractability, when describing and reasoning about component interactions. The authors use a sugared subset of the π-calculus for describing *interaction patterns*: sets of interactions that describe the finite interactive behaviour that a component may (repeatedly) show to the external environment. In opposition to our work, patterns allow for describing finite interactions only. Canal *et al.* [6, 7] use *roles* for defining partial protocol specifications. Although roles may alleviate some of the computational complexity of the substitutability tests, they are still NP-hard. In this sense our more lightweight approach

represents an improvement, despite of losing some expressiveness.

Finally, Architectural Description Languages (ADLs) usually include the descriptions of the protocols that determine the access to the components they define. Many ADLs use standard notations for describing component interactions (such as CSP, CCS, or π-calculus), which allow the simulation of the application's behaviour, or the formal derivation of some of its safety or liveness properties. For instance, the Rapide ADL is well known as a rich pattern language for the simulation of architectural behaviour. Wright [1] uses CSP for specification, and as a consequence is supported by model-checking tools such as FDR. Darwin [16] and LEDA [7] are examples of ADLs that make use of the π-calculus for describing the behaviour of the components of a system. One of the benefits of using standard calculi is that reasoning about the system behaviour and correctness can be done using appropriate tools. Our focus is slightly different, since we are more concerned with the specification of COTS components independently from the applications they will be part of. What we have shown here is that we can achieve the similar tests to those carried out by software architects with their ADLs (such as those described in [8, 9]), right from the objects' protocol specifications. In addition, our proposal not only contemplates compatibility tests but it also studies substitutability between components. Another novelty of our work is the dynamic tests that can be carried out with the interceptors, checking at run time that the behaviour of a CORBA object conforms to its declared protocol. They help detect incompatibility issues and possible violations to the protocols.

7. Future work

Session types allow the description of pairwise interactions between components. There are however situations where dyadic sessions are not expressive enough, where information on how separate sessions interleave is needed. This fact has surfaced, for instance, in work-flow applications, for which the order among the various events in sessions is critical. Another concern related to the information conveyed by session types is the impossibility of proving particular global properties of applications, such as absence of deadlocks among three or more partners. Session types deal with pairwise interactions, so they only allow to prove that local, dyadic, interactions are error-free.

The search for alternative uses of session types to describe protocols with multi-party interactions that allow useful compatibility and substitutability tests is currently under way. We are working on *Multi-Party Session Types* (MPST), an extension that allows the specification interaction among more than two components. With MPST we also help in solving the common trade-off between incorporating all the component interactions into one large protocol description (hence producing unwieldy and complex protocols), and breaking it into unrelated pairwise sessions (thus loosing information on how separate sessions interleave). MPST permits mixing protocols according to the system's particular requirements, allowing the component specifier to decide the level of interaction described by each session.

Acknowledgements. This work was partially supported by Portuguese-Spanish Bilateral Interchange (E15/02), by the Portuguese Fundação para a Ciência e a Tecnologia and by the EU FEDER (via CLC and the project MIMO, POSI/CHS/39789/ 2001), and by the EU IST FET-Global Computing (projects Mikado, IST–2001–32222, and Profundis, IST–2001–33100).

References

[1] Allen, R., Garlan, D.: A Formal Basis for Architectural Connection, *ACM Trans. on Software Engineering and Methodology*, **6**(3), July 1997, 213–249.

[2] Bastide, R., Sy, O.: Towards Components that Plug AND Play, *Proc. of the ECOOP 2000 Workshop on Object Interoperability (WOI'00)* (A. Vallecillo, J. Hernández, J. M. Troya, Eds.), June 2000.

[3] Bastide, R., Sy, O., Palanque, P.: Formal Specification and Prototyping of CORBA Systems, in: *Proceedings of ECOOP'99*, number 1628 in LNCS, Springer-Verlag, 1999, 474–494.

[4] Bracciali, A., Brogi, A., Turini, F.: Coordinating Interaction Patterns, *Proceedings of SAC'01*, ACM Press, October 2001.

[5] Canal, C., Fuentes, L., Pimentel, E., Troya, J. M., Vallecillo, A.: Extending CORBA Interfaces with Protocols, *The Computer Journal*, **44**(5), October 2001, 448–462.

[6] Canal, C., Fuentes, L., Pimentel, E., Troya, J. M., Vallecillo, A.: Adding Roles to CORBA Objects, *IEEE Transactions on Software Engineering*, **29**(2), February 2003, 242–260.

[7] Canal, C., Pimentel, E., Troya, J. M.: Compatibility and Inheritance in Software Architectures, *Science of Computer Programming*, **41**, 2001, 105–138.

[8] Compare, D., Inverardi, P., Wolf, A. L.: Uncovering architectural mismatch in component behavior, *Science of Computer Programming*, **33**(2), February 1999, 101–131, ISSN 0167-6423.

[9] Garlan, D., Allen, R., Ockerbloom, J.: Architectural Mismatch: Why Reuse is So Hard, *IEEE Software*, **12**(6), November 1995, 17–26.

[10] Gay, S., Hole, M.: Subtyping Session Types in the π-calculus, To appear.

[11] Gay, S., Hole, M.: *Types and Subtypes for Correct Communications in Client-Server Systems*, Technical report TR-2003-131, Department of Computing Science, University of Glasgow, 2003, An extended abstract appeared in *Proceedings of ESOP'99*, volume LNCS 1576, pages 74–90.

[12] Honda, K.: Types for Dyadic Interaction, *CONCUR'93*, 715, Springer-Verlag, 1993.

[13] Honda, K., Vasconcelos, V. T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming, *ESOP'98*, 1381, Springer-Verlag, 1998.

[14] Lea, D.: Interface-Based Protocol Specification of Open Systems using PSL, in: *Proc. of ECOOP'95*, number 1241 in LNCS, Springer-Verlag, 1995.

[15] Leavens, G. T., Sitaraman, M., Eds.: *Foundations of Component-Based Systems*, Cambridge University Press, 2000.

[16] Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour Analysis of Software Architectures, in: *Software Architecture*, Kluwer Academic Publishers, 1999, 35–49.

[17] Nierstrasz, O.: Regular Types for Active Objects, *Object-Oriented Software Composition* (O. Nierstrasz, D. Tsichritzis, Eds.), Prentice-Hall, 1995.

[18] OMG: *The Common Object Request Broker: Architecture and Specification*, Object Management Group, 2.4 edition, November 2000, `http://www.omg.org/technology/documents/formal/corbaiiop.htm`.

[19] Takeuchi, K., Honda, K., Kubo, M.: An Interaction-Based Language and its Typing System, *PARLE '94* (C. Halatsis, D. G. Maritsas, G. Philokyprou, S. Theodoridis, Eds.), 817, Springer-Verlag, 1994.

[20] Vasconcelos, V. T., Ravara, A., Gay, S.: Session types for functional multithreading, *CONCUR'04*, 3170, Springer-Verlag, 2004.

[21] Wehrheim, H.: Behavioural Subtyping Relations for Active Objects, *Formal Methods in System Design*, **23**, 2003, 143–170.

[22] Yellin, D. M., Strom, R. E.: Protocol Specifications and Components Adaptors, *ACM Transactions on Programming Languages and Systems*, **19**(2), March 1997, 292–333.

[23] Zaremski, A. M., Wing, J. M.: Specification Matching of Software Components, *ACM Trans. on Software Engineering and Methodology*, **6**(4), October 1997, 333–369.