Fundamenta Informaticae XX (2002) 1–15
IOS Press

Typing the Behavior of Objects and Components using Session Types

Antonio Vallecillo
Dep. of Computer Science, University of Málaga

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

António Ravara
Dep. of Mathematics, Instituto Superior Técnico, Technical University of Lisbon

Abstract. This paper describes a proposal for typing the behavior of objects in component models. Most component models, CORBA in particular, do not offer any support for expressing behavior al properties of objects beyond the "static" information provided by IDLs. We build on the works by Honda *et al*. [6] and Gay and Hole [5] to show how *session types* can be effectively used for describing protocols, extending the information currently provided by object interfaces. We show how session types not only allow high level specifications of complex object interactions, but also allow the definition of powerful interoperability tests at the protocol level, namely compatibility and substitutability of objects

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 people, 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

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

software components play a critical role in CBSD, since we need to be able to check whether a given component can successfully replace another in a particular application, or whether the behavior of two components is compatible for them to interoperate.

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 object and component platforms (such as CORBA, DCOM, or EJB) 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 this kind of (*signature*) interoperability is not sufficient for ensuring the correct development of component-based applications in open systems [14].

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 [8] for a comprehensive survey on these proposals). However, these proposals share a common drawback: the (computational) complexity of proving some of the behavioral properties of components and applications based on their full semantical descriptions hinders their practical utility.

Apart from signatures and behavioral semantics, another possibility is to concentrate just on the components interactions with other components, defining their service access protocols, and the way they use other components' services. This approach provides more than just signature information, it also allows the definition of compatibility and substitutability checks among components, and at a lower computational cost than other semantic tests.

Some authors have dealt with component interoperability at this level [3, 7, 13]—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 behavior 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 still very 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* [6] for describing the dynamic behavior of components. Sessions are partial protocol specifications, in which we only pay attention to the behavioral interface that a component presents to another one. This allows modular specification of the behavior of the components, providing more than just signature information, and permits precise definitions of compatibility and substitutability tests, and at a lower computational cost than other semantic checks (and hence of practical utility).

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 sessions, thus allowing powerful compatibility tests between components. The typability of a program ensures that two possibly communicating components always own compatible communication patterns. Moreover, typability also permits

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

Figure 1. A grammar for describing protocols

the definition of component substitutability checks based on the concept of session subtyping, which are computationally tractable in most cases, in contrast to the exponential (non-tractable) tests that result from the use of traces or process algebras in protocol descriptions.

Our work builds on the work by Honda *et al*. [6], which initially introduced session types for describing object service protocols. Protocol compatibility and substitutability tests are defined using the subtyping relation defined by Gay and Hole for session types [5]. In this paper we first complement those works by introducing the notion of *compatibility* between session types, extend it to objects, 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.

The structure of this paper is as follows. After this introduction, Section 2 introduces the language we propose for describing object protocol interactions, using an example application that will be used throughout the paper to illustrate our proposal. Section 3 introduces the type discipline supporting the language, including a subtyping relation '$\leq$' for session types via a proof system. This operator will 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. Finally, Section 5 relates our work to other similar approaches and draws some conclusions.

2. Expressing objects interactions via protocols

This section presents a language to describe protocols. The language is generated by the grammar in Figure 1.

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 (Figure 2).

The protocol with a seller 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 modelled by the invocation of operation sold on the seller; the second by the operation of operation notSold. In either case the protocols halts, as indicated by the **end** mark. The distinction between the inbound operation &{selling:...}, and the outbound operation +{sold:... |

```
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 2. Distributed auction bidding

notSold:...} must be stressed: the former denotes an operation provided by the auctioneer and invoked by any seller, the latter describes an operation of a seller invoked by an auctioneer.

The 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 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: 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 acknowledgements. 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 acknowledgements takes the protocol to the unregistering phase and then to halt. Session identifiers serve two purposes: recursive definitions (Bidding), and code structuring (Unregistering).

Consider now the protocol for a potential seller: it must provide an auctioneer with the description of the item to be sold (a **string**), and its minimum price (a **float**). Then it waits for 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 an operation).

As a last example consider a more apt seller, that upon emitting a selling order, is able to process three kinds of requests: the familiar sold/notSold, as well as the new lowerYourPrice, to which the seller 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 behavior 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 less-or-equal selections (+), and more-or-equal branchings (&); session SuperSeller::withAnAuctioneer is a *supertype* of session Seller::withAnAuctioneer: we write Seller::withAnAuctioneer $\leq$ SuperSeller::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 supertype 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 behavior never gets excited by the basic Auctioneer.

3. A type discipline for sessions

This section presents the the subtyping relation for session types via a proof system. As usual, T is a subtype of S, written $T \leq S$, if T can be used in any context where S is used and no error occurs in the session. Therefore, T should have more-or-equal branchings ($+$) and less-or-equal selections ($\&$). Based on this relation we say that T is compatible with S, $T \bowtie S$, if T is a subtype of a dual of S.

Recursive session types. For technical convenience, the language of types used in this section uses recursive definitions rather than equations. The grammar for types T in Figure 1 is enriched with a new production $\mu X.T$. If D is the set of equations $\{X_0 = T_0, \ldots, X_n = T_n\}$ taken from a given protocol, and I is the index set $\{0, \ldots, n\}$, then each type T_k (for $1 \leq k \leq n$) is translated into a recursive type accordingly to the following rule.

$$[\![T_k, D]\!] \stackrel{\text{def}}{=} T_k[\mu X_i.(T_i[\mu X_j.T_j/T_j]_{j \in I \setminus \{i\}})/X_i]_{i \in I}$$

$$\frac{T \,\mathsf{dualof}\, S \in \Sigma}{\Sigma \vdash T \,\mathsf{dualof}\, S} \quad \text{(D-ASSUMP)}$$

$$\Sigma \vdash \mathbf{end} \,\mathsf{dualof}\, \mathbf{end} \quad \text{(D-END)}$$

$$\frac{T \,\mathsf{dualof}\, S}{\Sigma \vdash ?(\widetilde{\text{sort}}); T \,\mathsf{dualof}\, !\widetilde{[\text{sort}]}; S} \quad \text{(D-SORT-IN)}$$

$$\frac{T \,\mathsf{dualof}\, S}{\Sigma \vdash !\widetilde{[\text{sort}]}; T \,\mathsf{dualof}\, ?(\widetilde{\text{sort}}); S} \quad \text{(D-SORT-OUT)}$$

$$\frac{T\tilde{T} \,\mathsf{dualof}\, S\tilde{S}}{\Sigma \vdash ?(\tilde{T}); T \,\mathsf{dualof}\, ![\tilde{S}]; S} \quad \text{(D-TYPE-IN)}$$

$$\frac{T\tilde{T} \,\mathsf{dualof}\, S\tilde{S}}{\Sigma \vdash ![\tilde{T}]; T \,\mathsf{dualof}\, ?(\tilde{S}); S} \quad \text{(D-TYPE-OUT)}$$

$$\frac{\Sigma \vdash T_i \,\mathsf{dualof}\, S_i \qquad \forall i \in \{1, \ldots, n\}}{\Sigma \vdash \&\{m_1 : T_1 \mid \cdots \mid m_n : T_n\} \,\mathsf{dualof}\, +\{m_1 : S_1 \mid \cdots \mid m_n : S_n\}} \quad \text{(D-BRANCH)}$$

$$\frac{\Sigma \vdash T_i \,\mathsf{dualof}\, S_i \qquad \forall i \in \{1, \ldots, n\}}{\Sigma \vdash +\{m_1 : T_1 \mid \cdots \mid m_n : T_n\} \,\mathsf{dualof}\, \&\{m_1 : S_1 \mid \cdots \mid m_n : S_n\}} \quad \text{(D-SELECT)}$$

$$\frac{\Sigma, \mu X.T \,\mathsf{dualof}\, S \vdash \mathsf{unwind}(\mu X.T) \,\mathsf{dualof}\, S}{\Sigma \vdash \mu X.T \,\mathsf{dualof}\, S} \quad \text{(D-REC-L)}$$

$$\frac{\Sigma, S \,\mathsf{dualof}\, \mu X.T \vdash S \,\mathsf{dualof}\, \mathsf{unwind}(\mu X.T)}{\Sigma \vdash S \,\mathsf{dualof}\, \mu X.T} \quad \text{(D-REC-R)}$$

Figure 3. Duality system

Type duality. In Section 2 we have hinted 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 (?). The actual definition is slightly complicated by the presence of recursive types. The inference rules are of the form $\Sigma \vdash T \,\mathsf{dualof}\, S$ where Σ is finite set of pairs of the form $T \,\mathsf{dualof}\, S$, meaning that type T is a dual of type S, assuming the pairs in Σ. The rules in Figure 3 inductively define a proof system for type duality, and allow to show, for example, that:

$$\vdash \mu X.?(\textbf{boolean}); X \;\; \mathsf{dualof} \;\; ![\textbf{boolean}]; \mu Y.![\textbf{boolean}]; Y$$

The example shows that the dual of a type is not unique; the "straightforward" dual $\mu X.![\textbf{boolean}]; X$ of the type $\mu X.?(\textbf{boolean}); X$ is another example.

Subtyping. Gay and Hole formally defined a subtyping relation for session types by means of a collection of inference rules for judgements of the form $\Sigma \vdash T \leq S$, where Σ is finite set of inequalities $T \leq S$, meaning that type T is a supertype of type S, assuming the inequalities in Σ [5]. When $\emptyset \vdash T \leq S$ is derivable, we will simply write $T \leq S$. Using this operator, in our context the expression '$T \leq S$' will (indistinctly) mean: "*T is more restrictive than S*"; "*S is more general than T*"; "*T can (safely) replace S*"; "*S is substitutable by T*"; or "*T is a subtype of S*".

The subset of the rules proposed by Gay and Hole that we need for subtyping session types is in Figure 4, where $\mathsf{unwind}(\mu X.T) \stackrel{\text{def}}{=} T[\mu X.T/X]$. These rules inductively define a proof system for subtyping. Based on this relation, we are finally in a position to define the concepts of substitutability and compatibility between session types.

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 $T \leq U$, for some $U \,\mathsf{dualof}\, S$.

Example. Let us show that session $S \stackrel{\text{def}}{=}$ Seller::withAnAuctioneer can safely substitute session $T \stackrel{\text{def}}{=}$ SuperSeller::withAnAuctioneer and that session T is compatible with session $U \stackrel{\text{def}}{=}$ Auctioneer::withASeller.

1. Recall that
   $S \stackrel{\text{def}}{=} +\{$ selling: ![**string**, **float**]; $S' \}$
   where
   $S' \stackrel{\text{def}}{=} \&\{$ sold: ?(**float**); **end** | notSold: **end** $\}$,
   and that
   $T \stackrel{\text{def}}{=} +\{$ selling: ![**string**, **float**]; $T' \}$
   where
   $T' \stackrel{\text{def}}{=} \mu X.\&\{$ sold: ?(**float**); **end** | notSold: **end** | lowerYourPrice: $T'' \}$
   with

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

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

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

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

$$\frac{\Sigma \vdash T \leq S \qquad \Sigma \vdash T_i \leq S_i \qquad \forall i \in \{1, \ldots, n\}}{\Sigma \vdash ?(\tilde{T}); T \leq ?(\tilde{S}); S} \quad \text{(S-TYPE-IN)}$$

$$\frac{\Sigma \vdash T \leq S \qquad \Sigma \vdash S_i \leq T_i \qquad \forall i \in \{1, \ldots, n\}}{\Sigma \vdash ![\tilde{T}]; T \leq ![\tilde{S}]; S} \quad \text{(S-TYPE-OUT)}$$

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

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

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

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

Figure 4. Subtyping system

$T'' \stackrel{\text{def}}{=} +\{$ ok: ![**float**]; X | noWay: **end** $\}$.

To show that $S \leq T$, apply first the rule S-BRANCH to conclude $S' \leq \mathsf{unwind}(T')$, then the rule S-REC-R to conclude $S' \leq T'$, and finally the rule S-SELECT.

2. Recall that

$U \stackrel{\text{def}}{=} \&\{$ selling: ?(**string**, **float**); $U' \}$

where

$U' \stackrel{\text{def}}{=} +\{$ sold: ![**float**]; **end** | notSold: **end** $\}$.

To show that $U \bowtie T$ one must find V such that V dualof U and $V \leq T$. It is simple to check that session S is the V that one needs: using rules D-TYPE-IN and D-BRANCH one concludes that S' dualof U'; by rules D-TYPE-OUT and D-SELECT one reaches the desired conclusion. The result follows by proving that $S \leq T$, what we have already done in the previous item.

Results. The following results show that substitutability is well defined, in the sense that subtyping is a preorder, and stress some properties of the duality and the compatibility relations. Note that compatibility is a symmetric relation, but is neither reflexive nor transitive, since duality is obviously not reflexive, but being symmetric is thus not transitive. The most interesting result is that if a session S is a subtype of some session T, then a dual of T is a subtype of a dual of S.

Proposition 3.1.

1. The relation dualof is symmetric.
2. Subtyping $\leq$ is a preorder.
3. $S \leq T$ if and only if $V \leq U$, for all U dualof S and all V dualof T.
4. Compatibility $\bowtie$ is symmetric.
5. If $U \leq T$ and $T \bowtie S$, then $U \bowtie S$.

Proof:
The proof of the first clause is by a simple induction on the derivation of the judgement S dualof T. The proof of the second clause is a straightforward consequence of the definition of $\leq$. To prove the third clause it suffices to show the 'only-if' direction; then the 'if' direction follows from clause one. For the 'only-if' direction, proceed by induction on the derivation of the judgement $V \leq U$. The fourth clause is a direct consequence of clause 3 and of the definition of '$\bowtie$'. Finally, the fifth clause is a consequence of the transitivity of $\leq$. □

Checking components. Given the relations of substitutability and compatibility of sessions, it is easy to check whether two components are substitutable (or compatible): one simply has to check that each session in the protocol of one of the components is substitutable for (or compatible with) some session in the protocol of the other component. To talk about which session relates to which session within two given protocols, we introduce the notion *set of bindings*, a set of pairs of session identifiers.

Definition 3.2. Let $\{X_1 = T_1, \ldots, X_n = T_n\}$ and $\{Y_1 = T_1, \ldots, Y_m = T_m\}$ be two sets D and D' of equations extracted from two given protocols P and Q, and let $\{(X_1, Y_1), \ldots, (X_k, Y_k)\}$ be a set B of bindings, where $k \leq n$ and $k \leq m$. We say that the two protocols P and Q are:

1. *substitutable over the set B of bindings*, if $[\![T_i, D]\!] \leq [\![T'_i, D']\!]$, forall $1 \leq i \leq k$;
2. *compatible over the set B of bindings*, if $[\![T_i, D]\!] \bowtie [\![T'_i, D']\!]$, forall $1 \leq i \leq k$.

It is now easy to conclude that SuperSeller can safely substitute Seller, and that SuperSeller is compatible with Auctioneer; just take for B the set $\{(\mathsf{withAnAuctioneer}, \mathsf{withASeller})\}$ in both cases.

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 (`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 [11].

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 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 set of types, attributes and the signature (parameters, return types and exceptions raised) of the object methods, grouped into **interface** definitions. Finally, the construct **module** is used to hold type definitions, interfaces, and other modules for name scoping purposes. As an example, let us describe a simple CORBA object given by the following interface:

```
interface Bidder {
  boolean wannaBid (in string itemDesc, in float price);
  void youGotIt (in string itemDesc, in float price);
  void itemSold (in string itemDesc);
}
```

It corresponds to the Bidder object described in Section 2. As we can see, the CORBA IDL allows us to describe the signature of the operations implemented by an object, but it does not allow the description of the external operations *required* by that object, or the partial order (i.e. the *protocol*) in which both the provided and used operations are expected to be invoked.

Expressing CORBA objects interactions We now concentrate on how to add protocol information to the description of the CORBA object interfaces, using the language constructs defined in Section 2.

CORBA objects' *protocols* are defined by two sets of interfaces and a set of session types. The first set describes the set of *provided* CORBA interfaces that the component supports (i.e. implements), each one under a **provides** heading. Second, we have the set 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 object 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 set of session types, each of them indicated by a **session** clause. The first two sets contain information at the

signature level only, while the last one is in charge of specifying the dynamic aspects of the behavior of the object. The grammar for describing *CORBA protocols* is obtained from that in Figure 1 by replacing the first production by the one below

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

The main modelling techniques that we propose for describing CORBA object interactions are 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 all **in** and **inout** arguments are sent in the ![...] output action, written in the same order they were declared in the IDL. In the method response (i.e. the ?(...) input action) 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 1.

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

4. Invocation of method "s m(s$_1$,...,s$_k$)" is modelled inside a *select* (+{...}) structure as "m:![s$_1$,...,s$_k$];?(s);".

5. Analogously, acceptance (and reply) of method "s m(s$_1$,...,s$_k$)" is modelled by "m:?(s$_1$,...,s$_k$);![s];" inside a *branch* (&{...}) structure.

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, the protocol that defines the dynamic behavior of object Bidder is shown in Figure 5, where objects Bidder make use of the services provided by an Auctioneer object, whose interface is the following:

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

It is important to note that session types describing CORBA interactions follow a reduced set of standard communication patterns, which are given by a subset of the full grammar described in Figure 1. This is due to the fact that CORBA imposes some restrictions on the communication patterns used by its objects, since client-server method invocation is the only mechanism allowed. Thus, a server may just offer several of its methods within a branch structure, then accept its 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

```
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 object protocol.

arbitrary manner. And likewise for the client. This fact facilitates both the description of the CORBA object protocols, and the way to reason about their interactions.

Another issue worth noticing is the need to accommodate to the CORBA specific way of working when describing CORBA object protocols with the constructs defined in Section 2. For instance, comparing the protocols in Figures 2 and 5 we can see that CORBA methods return a value (therefore the ![**void**] not present in Figure 2), and that all arguments appearing in the CORBA IDL interfaces must be present in the protocol description (eg. the Bidder parameter of method register). Although not needed in generic protocols, they are necessary when describing the behavior of the CORBA objects implementing a particular CORBA interface.

5. Further issues and related work

The work we describe in this paper is still preliminary. This section discusses several issues that need to be further investigated, and compare our approach to other related proposals.

Testing compatibility. The notions substitutability and compatibility are based those of subtyping and duality. The two notions are decidable; there are algorithms for checking whether $T \leq S$ and whether T dualof S. A simple algorithm to check subtyping is presented by Gay and Hole [5]; from that one can easily imagine one for checking duality. To check the compatibility of types T, S, we build a "straightforward" dual $\overline{S}$ of S (see the definition in Honda *et al.* [6] or in Gay and Hole), and then check the subtype relation $T \leq \overline{S}$. Notice that our proof system for duality (Section 3) is correct with respect to "straightforward duals". We plan to study the computational complexity of these algorithms, although we expect it to be tractable.

Static type conformance. We would like to be able to statically check that a given protocol implementation conforms to a given protocol interface that supposedly describes its behaviour. While that is certainly easy to do with message-passing-based languages (π-based languages, for example [5, 6]), the scenario is not so bright when it comes to languages whose communication mechanism is not based on message-passing. Transferring session types from the π-calculus to an imperative or object-oriented language is a open and challenging problem.

Dynamic type conformance. While we do not know how to statically check type conformance, or when we do not have access to the source code, we can dynamically check type conformance, that is, check that an actual protocol implementation does conform to a given type, during execution. The idea is to set up an *interceptor* that monitors all incoming and outgoing messages to and from an object. It should be easy to build an interceptor for a particular session type as an automata whose arcs are labelled with the constructors of types T in Figure 1 ($\&m; +m, ![\tilde{T}], ?(\tilde{T}), ![\widetilde{\text{sort}}]$, and $?(\widetilde{\text{sort}})$).

CORBA in particular provides this kind of entities, by means of *interceptor* objects [11], which are local objects that can be attached to any CORBA object, acting as filters that intercept and observe all the object's incoming and outgoing messages. Thus, they allow a programmer to specify additional code to be executed before or after the normal code of an operation. This code can be used for observing the messages exchanged among the components of an application, checking whether they conform to a valid predetermined behaviour. Therefore, an interceptor can be defined for each object, and attached to it when created. Invalid messages with regard to a given protocol can be captured and handled as required.

Programming protocols in imperative languages. Close to the problem of static type conformance above lies that of programming protocols in imperative (in particular concurrent object-oriented) languages. One could create a different object for each step of a protocol, and send on, each method invocation, a reference for the object that continues the protocol. Output following a selection (+{ sold: ![float]. . .), a quite common pattern, is implemented as a method call and its arguments; input following branching (&{ selling: ?(**string**, **float**). . .) is implemented as method declaration and its parameters. For example protocol Seller in Section 2 could written in Java as

anAuctioneer.selling ("my car", 1500.0, new SoldNotSold ()),

where SoldNotSold is a class with two methods (sold, notSold), whose instances implement the continuation of the protocol. As already mentioned in [6], the resulting program is hard to understand, and feels unnatural. However, the code may be simpler and clearer adopting a "behaviour-oriented" style of programming like in non-uniform objects—objects that may dynamically change their behaviour and even the methods they offer—as advocated by Nierstrasz [10] and studied by Ravara [12].

Related work. As mentioned in the introduction, several authors have provided a number of proposals that try to overcome the limitations that current component IDLs present, defining extensions that cope with the semantic aspects of object interfaces and behavior. Apart from those that try to deal with the full operational semantics of components (discussed in [8]), there are several proposals that cover the specification of the objects' service protocols using different notations—from finite state machines to process algebras [1, 3, 7, 10, 13]. However, all these proposals 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 object interactions can be expressed in terms of session types, not full-blown interaction protocols.

An interesting proposal by Braciali *et al.* [2] also sacrifices expressiveness in order to achieve modularity and computational tractability when describing and reasoning about component interactions. The authors use a sugared subset 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 their approach, interaction patterns do not contain recursion, hence projecting the full-blown behavior of a component over 'time', instead of over 'space' as we do in our work. Projecting over space is also the approach used by Canal *et al.* [4], that 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 (sessions, unlike roles, can only describe pairwise interactions).

Finally, some Architectural Description Languages (ADLs) also include the descriptions of the protocols that determine the access to the components they define using standard notations that derive from process algebras (like CSP, CCS or π-calculus). One of the benefits of using standard calculi is that reasoning about system behavior and correctness can be done using appropriate tools. Darwin [9] and LEDA [4] are examples of ADLs that make use of the π-calculus for describing the behavior of the components of a system. Our focus is somehow different, since we are more concerned with the specification of software components, independently from the applications they will be part of. However, what we have shown here is that we can achieve the same sort of tests that software architects carry out with their ALDs, right from the objects' protocol specifications.

Acknowledgements. This work was partially supported by Portuguese-Spanish Bilateral Interchange (E15/02), by the Portuguese Foundation for Science and Technology (project MIMO, POSI/CHS/39789/2001), and by EU IST FET-Global Computing (project Mikado, IST–2001–32222).

References

[1] 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.

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

[3] 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.

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

[5] Gay, S. J., Hole, M.: *Types for Correct Communication in Client-Server Systems*, Technical report, Department of Computer Science, Royal Holloway, University of London, 2000, Extended version of Types and Subtypes for Client-Server Interactions, in Proceedings of the European Symposium on Programming Languages and Systems, Springer-Verlag LNCS 1576, 1999.

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

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

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

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

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

[11] 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`.

[12] Ravara, A.: *Typing non-uniform concurrent objects*, Ph.D. Thesis, Technical University of Lisbon, Portugal, 2000.

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

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