

# Behavioural Types and Component Adaptation <sup>★</sup>

Antonio Brogi<sup>1</sup>, Carlos Canal<sup>2</sup>, and Ernesto Pimentel<sup>2</sup>

<sup>1</sup> Department of Computer Science, University of Pisa, Italy  
brogi@di.unipi.it

<sup>2</sup> Department of Computer Science, University of Málaga, Spain  
{canal, ernesto}@lcc.uma.es

**Abstract.** Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering. The objective of this paper is to set a formal foundation for the adaptation of heterogeneous components that present mismatching interaction behaviour. The proposed adaptation methodology relies on: (1) the inclusion of behavioural types in component interfaces, to describe the interaction behaviour of components, (2) a simple high-level notation for adaptor specification, to express the intended connection between component interfaces, and (3) a formal definition of adaptor, a component-in-the-middle capable of making two components interoperate successfully according to a given specification.

## 1 Introduction

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering [4, 5, 10]. The capability of adapting off-the-shelf software components to work properly within larger applications is a must for the development of a true component marketplace, and for component deployment in general [3]. The need for component adaptation is also motivated by the ever-increasing attention devoted to developing extensively interacting distributed systems, consisting of large numbers of heterogeneous components.

Available component-oriented platforms address software interoperability at the signature level by means of Interface Description Languages (IDLs), a sort of *lingua franca* for specifying the functionality offered by heterogeneous components. While IDLs allow to overcome signature mismatches, there is no guarantee that the components will interoperate correctly, as mismatches may occur because of differences in the interaction behaviour of the components involved [11].

The objective of this paper is to set a formal foundation for the *adaptation* of heterogeneous components that may present mismatching interaction behaviour. The notion of *adaptor* was originally introduced in [13] to name a component-in-the-middle aimed at enabling the successful interoperation of two components presenting mismatching behaviour.

---

<sup>★</sup> This work has been partly supported by the project NAPOLI funded by the Italian Ministry of Instruction, University and Research (MIUR), and the projects TIC2002-4309-C02-02 and TIC2001-2705-C03-02 funded by the Spanish Ministry of Science and Technology (MCYT).

In our previous work [2], we have developed a formal methodology for component adaptation that supports the successful interoperation of heterogeneous components presenting mismatching interaction behaviour. As pointed out in [13], the first step needed to overcome behavioural mismatch is to let behaviour information be explicitly represented in **component interfaces**. Process algebras feature a very expressive description of interaction protocols, and enable sophisticated analyses of concurrent systems. For these reasons, their use for the specification of component interfaces and for the analysis of component compatibility has been widely advocated (e.g. [1, 7, 9]). On the other hand, a serious drawback of employing process algebras is the inherent complexity of verification procedures, which inhibits their usability in practice.

Therefore, a suitable trade-off between expressiveness and effectiveness of verification is needed to reason about component protocols. To this end, we propose the notion of *session* for the description of component interfaces. Intuitively speaking, sessions feature a modular projection of component behaviour both in space and in time. On the one hand, each session describes a partial view of the behaviour of a component (w.r.t. another component that will be attached to it), thus partitioning the component behaviour into several facets or roles. On the other hand, each session describes a (possibly finite) connection, thus partitioning the full life of a component into a sequence of sessions.

From a technical viewpoint, we will use *session types* (firstly defined in [6]) to describe component sessions as true types. The ultimate objective of employing session types is to provide a basic means to describe complex interaction behaviour with clarity and discipline at a high-level of abstraction, together with a formal basis for analysis and verification. Session types are supported by a rigorous type discipline, thus featuring a powerful type checking mechanisms of component behaviour. Moreover, the use of types —instead of processes— to describe behaviour features the possibility of describing recursive behaviour while maintaining the analysis tractable.

The second ingredient necessary to address formally the issue of component adaptation is a suitable formalism to express **adaptor specifications**. Indeed, as shown in [2], separating adaptor specification and adaptor derivation permits the automation of the error-prone, time-consuming task of manually constructing a detailed implementation of a correct adaptor. The desired adaptation will be expressed by simply defining a set of (possibly non-deterministic) correspondences between the actions provided by the two components to be adapted. As we will see, the distinguishing aspect of the notation used is that it produces a high-level, partial specification of the adaptor.

In a third step, we define formally the notion of **adaptor** in terms of an adaptor specification and of the session types of the components involved. Finally, we prove that adaptors guarantee the safe interaction of the components adapted, in the sense that they will never deadlock during an interaction session.

The rest of the paper is organized as follows. Sect. 2 is devoted to introduce session types for typing component behaviour. After defining a process calculus to denote component protocols, a type system is introduced and the notion of

type compatibility is presented. In Sect. 3 adaptor specifications and the formal definition of adaptor are introduced, and the session-safety result is established. Finally some concluding remarks are drawn in Sect. 4.

## 2 Typing component behaviour

Process algebras have been widely used to specify software systems. In particular, they have been often employed to describe the interactive behaviour of software components [1, 7, 9]. The advantage of having these formal specifications of components is two-fold. First, component behaviour is unambiguously fixed and it can be animated with a convenient interpreter tool. Second, it is possible to analyze a number of liveness and security properties such as safe composition or replaceability in complex systems. In spite of the usefulness of process algebras for component description, they present an important drawback due to the complexity of the decision procedures to verify the mentioned properties. In order to cut off this complexity, we have applied to our context the notion of *session type* introduced in [6].

Session types present some important features that distinguish them from processes written in a general process algebra like the  $\pi$ -calculus:

- session types abstract from data values, referring to the corresponding data types instead;
- sessions are limited to diadic communications between two components;
- mobility is expressed by means of explicit *throw/catch* actions, and since sessions are diadic, once a process throws a session, it cannot use it anymore;
- no mixed alternatives are allowed: input and output actions cannot be combined in a non-deterministic choice.

It is worth noting that these restrictions are not relevant limitations in our context, as we will show. On the contrary, they make session types a calculus much more tractable than other studied alternatives, like the  $\pi$ -calculus or CSP. A thorough discussion of the advantages of employing session types instead of other concurrency formalisms is reported in [12]. Under this approach, a process is viewed as a collection of sessions, each one being a chain of diadic interactions. Each session is designated by a private session name, through which interactions belonging to that session are performed. The use of diadic sessions for the specification of software interaction allows a modular description of complex systems. The objective is to provide a basic means to specify complex interaction behaviour with clarity and discipline at a high-level of abstraction, together with a formal basis for analysis and verification.

Throughout the paper, we will use both a session type description language and a process calculus  $\mathcal{L}$ . The former will be used to type the behaviour of components (and will be exposed in component interfaces), while the latter will be used to refer to (and exemplify) the actual implementation of components.

## 2.1 A process calculus for component description

In this section we present the process calculus for describing component implementations. It is a variant of that used in [6]. Apart from some simplifications in the notation, the main difference is that we allow the alternative composition of output actions (somehow equivalent to an *if-then-else* construct), and not only of input actions as in [6]. We give also a transition system for the calculus, not present in [6]. The syntax of the process calculus  $\mathcal{L}$  is defined as follows:

$$P ::= 0 \mid act.P \mid \sum_i k!m_i.P_i \mid \sum_i k?m_i.P_i \mid P_1 \parallel P_2 \mid A(\tilde{x}\tilde{k})$$

$$act ::= x!request(k) \mid x?accept(k) \mid k!throw(k') \mid k?catch(k')$$

where  $0$  represents the empty process,  $P$ ,  $P_i$  denote a process,  $A$  is a process identifier,  $x$  denotes a link name,  $k$  and  $k'$  denote session names,  $\tilde{\cdot}$  denotes a sequence of names, and  $m_i$  denotes a message, syntactically composed by a message selector and a sequence of data arguments.

For any process identifier  $A(\tilde{x}\tilde{k})$  there must be a unique defining equation  $A(\tilde{x}\tilde{k}) = P$ . Then,  $A(\tilde{y}\tilde{h})$  behaves like  $P\{\tilde{y}/\tilde{x}, \tilde{h}/\tilde{k}\}$ . Defining equations provide recursion, since  $P$  may contain any process identifier, even  $A$  itself.

We consider two kinds of actions in the process calculus  $\mathcal{L}$ : output actions ( $k!m_i$ ), where a message  $m_i$  is sent through a session  $k$ , and input actions ( $k?m_i$ ), where a message  $m_i$  is received through a session  $k$ . There are four special message selectors: *request*, *accept*, *throw*, and *catch*. All of them require a single argument representing a session name. A *request* output action issued on a link name  $x$  waits for the acceptance (*accept*) of a session on this link. When these two actions synchronize, a new session is created linking the processes where the interaction was performed. Similarly, *throw* and *catch* are complementary actions, too. In this case, an existing session (name) can be moved from a process (where the *throw* action is made) to another one (where a *catch* action is performed to capture the session). These last two actions permit to change dynamically the topology of the system. Notice that link names are only used to create sessions (via *request/accept* actions), while all other interactions take place on session names.

The transition relation described in Fig. 1 defines the operational semantics of  $\mathcal{L}$ . There are both labelled and unlabelled transitions, the latter corresponding to silent actions. The four initial rules describe the behaviour of actions concerning session manipulations. Rules (REQ) and (ACC) model session creation, while (THR) and (CTH) model session transmission. Both *accept* and *catch* are binding actions, receiving a new session name not occurring in  $fn(P)$  (free names of  $P$ ). Rule (SUM) defines the behaviour of a sum of (either input or output) actions  $\lambda_i = k?m_i$  or  $\lambda_i = k!m_i$ , respectively, which is modelled by the usual non-deterministic choice, assuming that the choice is made locally for output actions, and globally for input actions. To model the parallel composition of processes we have four different transition rules. Rules (PAR<sub>open</sub>) and (PAR<sub>throw</sub>) model session opening and session throwing, respectively, whereas (SYNC) models

$$\begin{array}{l}
\text{REQ} : \frac{}{x!request(k).P \xrightarrow{x!rqt(k)} P} \quad \text{THR} : \frac{}{k!throw(k').P \xrightarrow{k!thw(k')} P} \\
\text{ACC} : \frac{}{x?accept(k).P \xrightarrow{x?acp(h)} P\{h/k\}} \quad (h \notin fn(P) \setminus \{k\}) \\
\text{CTH} : \frac{}{k?catch(k').P \xrightarrow{k?cth(h)} P\{h/k'\}} \quad (h \notin fn(P) \setminus \{k'\}) \\
\text{SUM} : \frac{\sum_{i=1}^n \lambda_i.P_i \xrightarrow{\lambda_j} P_j}{\sum_{i=1}^n \lambda_i.P_i \xrightarrow{\lambda_j} P_j} \quad (j = 1 \dots n) \\
\text{PAR}_{open} : \frac{P \xrightarrow{x!rqt(k)} P' \quad Q \xrightarrow{x?acp(k)} Q'}{P \parallel Q \xrightarrow{} P' \parallel Q'} \\
\text{PAR}_{throw} : \frac{P \xrightarrow{k!thw(k')} P' \quad Q \xrightarrow{k?cth(k')} Q'}{P \parallel Q \xrightarrow{} P' \parallel Q'} \\
\text{SYNC} : \frac{P \xrightarrow{k!m} P' \quad Q \xrightarrow{k?m} Q'}{P \parallel Q \xrightarrow{} P' \parallel Q'} \quad \text{PAR} : \frac{P \xrightarrow{\lambda} P'}{P \parallel Q \xrightarrow{\lambda} P' \parallel Q} \\
\text{DEF} : \frac{P\{\tilde{y}/\tilde{x}, \tilde{h}/\tilde{k}\} \xrightarrow{\lambda} P'}{A(\tilde{y}\tilde{h}) \xrightarrow{\lambda} P'} \quad (A(\tilde{x}\tilde{k}) = P)
\end{array}$$

**Fig. 1.** Transition system for  $\mathcal{L}$ .

the synchronous exchange of input and output messages. Rule (PAR) describes the standard interleaving of parallel compositions. Finally, rule (DEF) models process definition. Note that the label  $\lambda$  in these last two rules may be empty. Additionally to the transition system in Fig. 1, we assume also standard commutativity and associativity axioms for choice and parallel composition operators.

Throughout this paper we will use a simplified FTP system to illustrate the different aspects of our proposal. Suppose that the system is composed of a server, with whom clients interact for opening FTP sessions, and a set of  $n$  daemons, each one responsible for handling one client session:

$$System(client, daemon) = Server(client, daemon) \parallel \prod_{i=1}^n Daemon(daemon)$$

Suppose also that the specification in  $\mathcal{L}$  of the *Server* component is:

```

Server(client, daemon) =
  client?accept(a). a?user(usr). a?pass(pwd).
  ( a!rejected("User unauthorized"). Server(client, daemon)
    + a!rejected("Service unavailable"). Server(client, daemon)
    + a!connected!(). daemon!request(b). a!throw(b). Server(client, daemon) )

```

The FTP *Server* component declares two link names (*client* and *daemon*). The first one is used for its interaction with its clients, while the second is used for interacting with the daemons. When a request is received on link *client*, a new session *a* is created, for handling the interaction between the system and this

specific client. Then, the client identifies itself, providing its name and password. As a result, the server may either reject (message `rejected`), or accept (message `connected`) the connection. In the latter case, the server requests on link `daemon` a session `b` from one of its daemons, and throws the newly created session `b` to the client. Session `a` ends now, and the server returns to its original state, ready for attending a new client request.

On the other hand, the daemon component can be represented as follows:

```

Daemon(server) = server?accept(b). Daemon2(server,b)
Daemon2(server,b) = ( b?get(filename).
                      ( b!result(data). Daemon2(server,b)
                        + b!nosuchfile(). Daemon2(server,b) )
                      + b?quit(). Daemon(server) )

```

The specification above shows how once a new session `b` is established with the server, the daemon waits for different user commands (here, only `get` is considered, the rest being alike), finishing with a `quit` command, after which the session ends, and the daemon is ready for accepting a new session request. After each `get` command, either the corresponding data (`result`) or an error (`nosuchfile`) is returned. Notice how the daemon remains unaware of the fact that the session `b` initially requested by the server is afterwards sent to the client, which is the one who actually performs `get` and `quit` commands.

However, our interest is not focused on using a process calculus like  $\mathcal{L}$  for describing the behaviour of software components, but rather in *typing* this behaviour for establishing the correct interaction among the corresponding components. This is the objective of the next section.

## 2.2 Typing system

Whereas the type system defined in [6] deals both with data and session types, without loss of generality we shall omit data arguments in process definitions and message arguments. This simplification is not relevant, and our typing system could be easily extended to deal also with data. We will denote by  $TExp$  the set of type expressions constructed by following grammar:

$$\alpha ::= 0 \mid \perp \mid !\alpha \mid ?\alpha \mid !(\alpha).\beta \mid ?(\alpha).\beta \mid !\sum_i t_i.\alpha_i \mid ?\sum_i t_i.\alpha_i \mid \Lambda$$

where  $\alpha$ ,  $\alpha_i$ ,  $\beta$  are type expressions, and  $\Lambda$  is a type identifier (we assume that for each identifier  $\Lambda$  exists a unique defining equation  $\Lambda = \alpha$ ). The constant type  $0$  represents inaction's type, and  $\perp$  denotes a specific type indicating that no further connection is possible at a given session. In other words, if a session  $k$  has a type  $\perp$  in a process, then  $k$  is not offered by this process as an open session. Type expressions  $!\alpha$  and  $?\alpha$  correspond to *request* and *accept* primitives, respectively, whereas  $!(\alpha)$  and  $?(\alpha)$  correspond to *throw* and *catch*. The expression  $t_i$  denotes the type associated to a message (which will coincide with the message selector, since we abstract from message arguments). Then, the type  $?\sum_i t_i.\alpha_i$

denotes the branching behaviour given by a process which is waiting with several options, and which behaves as type  $\alpha_i$  if the  $i$ -th action is selected. Similarly,  $!\sum_i t_i.\alpha_i$  denotes the complementary selecting behaviour, w.r.t. output actions.

Given a type  $\alpha$  where  $\perp$  does not occur, we define its dual type  $\bar{\alpha}$ , as follows:

$$\begin{array}{l} \overline{?\alpha} = !\bar{\alpha} \quad \overline{?(\alpha).\beta} = !( \alpha ).\bar{\beta} \quad \overline{!\sum_i t_i.\alpha_i} = ?\sum_i t_i.\bar{\alpha}_i \quad \bar{0} = 0 \\ \overline{!\alpha} = ?\bar{\alpha} \quad \overline{!(\alpha).\beta} = ?(\alpha).\bar{\beta} \quad \overline{?\sum_i t_i.\alpha_i} = !\sum_i t_i.\bar{\alpha}_i \end{array}$$

The dual of a given type denotes the complementary behaviour of the original type, and  $\bar{\bar{\alpha}} = \alpha$ . It is obvious that the composition of a type and its dual is successful, in the sense that the corresponding processes will execute without deadlocks [6], eventually terminating in empty processes. However, imposing such a condition seems too restrictive in the context of real software components.

In [12] a notion of type compatibility is defined in terms of a subtyping relation and type duality. Intuitively speaking, a session type  $\alpha$  is a subtype of  $\beta$  if  $\alpha$  can be used in any context where  $\beta$  is used, and no error occurs in the session. Basically, this means that  $\alpha$  should have more —or equal— branchings (input alternatives), and less —or equal— selections (output alternatives). Based on this subtyping relation,  $\alpha$  is said compatible with  $\beta$ , denoted by  $\alpha \bowtie \beta$ , if  $\alpha$  is a subtype of the dual of  $\beta$ . Defined this way, compatibility is a sufficient condition to ensure successful composition of the corresponding processes. (More details about these subtyping and compatibility relationships can be found in [12].)

The typing system for the calculus  $\mathcal{L}$  is shown in Fig. 2, and deals with sequents of the form:  $\Theta; \Gamma \vdash P \triangleright \Delta$ , which means: “under the current environment, given by  $\Theta$  and  $\Gamma$ , the process  $P$  has a typing  $\Delta$ ”. As in [6],  $\Theta$  denotes a *basis*, which is a mapping from process names to the types in  $TExp$  of the corresponding arguments (links and sessions); while the *sorting*  $(t \in) \Gamma$  stores types for links, which are expressed by means of sorts  $\langle \alpha, \bar{\alpha} \rangle$ . A sort of this form represents a pair of complementary interactions which are associated with a link name: one starting with *accept*, and the other one starting with *request*. Each of them correspond to the type of certain session in the *typing*  $\Delta$ , which is a partial mapping from session names to types. Given a typing (or sorting or basis)  $\Xi$ , we write  $\Xi \cdot k : \alpha$  to denote the mapping  $\Xi \cup \{k : \alpha\}$  provided that  $k \notin \text{dom}(\Xi)$ .

Using session types for describing component behaviour makes it possible to determine when two components can interact safely. This analysis will be done in terms of the compatibility of the typings of the components. Indeed, the notion of compatibility in [12], previously mentioned, can be naturally extended to typings. When two typings,  $\Delta_1$  and  $\Delta_2$ , are compatible ( $\Delta_1 \bowtie \Delta_2$ ), their composition ( $\Delta_1 \odot \Delta_2$ ) is defined as a new typing given by:

$$(\Delta_1 \odot \Delta_2)(k) = \begin{cases} \perp & \text{if } k \in \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) \\ \Delta_1(k) & \text{if } k \in \text{dom}(\Delta_1) \setminus \text{dom}(\Delta_2) \\ \Delta_2(k) & \text{if } k \in \text{dom}(\Delta_2) \setminus \text{dom}(\Delta_1) \end{cases}$$

As we have already said, the typing system in Fig. 2 is similar to that provided in [6], but adapted to the process calculus  $\mathcal{L}$ . The first two rules define the sort associated to a link  $x$ , on which an *accept* or *request* is made. Notice that the sort

$$\begin{array}{c}
\mathcal{T}_{\text{ACC}} : \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta; \Gamma, x : \langle \alpha, \bar{\alpha} \rangle \vdash x? \text{accept}(k).P \triangleright \Delta \cdot k : ?\alpha} \\
\mathcal{T}_{\text{REQ}} : \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \bar{\alpha}}{\Theta; \Gamma, x : \langle \alpha, \bar{\alpha} \rangle \vdash x! \text{request}(k).P \triangleright \Delta \cdot k : !\alpha} \\
\mathcal{T}_{\text{THR}} : \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \beta}{\Theta; \Gamma \vdash k! \text{throw}(k').P \triangleright \Delta \cdot k : !(\alpha), \beta \cdot k' : \alpha} \\
\mathcal{T}_{\text{CTH}} : \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \beta \cdot k' : \alpha}{\Theta; \Gamma \vdash k? \text{catch}(k').P \triangleright \Delta \cdot k : ?(\alpha), \beta} \\
\mathcal{T}_{\text{IN}} : \frac{\Theta; \Gamma \vdash P_1 \triangleright \Delta \cdot k : \alpha_1 \quad \cdots \quad \Theta; \Gamma \vdash P_n \triangleright \Delta \cdot k : \alpha_n}{\Theta; \Gamma \vdash \sum_{i=1}^n k?m_i.P_i \triangleright \Delta \cdot k : ?\sum_{i=1}^n m_i.\alpha_i} \\
\mathcal{T}_{\text{OUT}} : \frac{\Theta; \Gamma \vdash P_1 \triangleright \Delta \cdot k : \alpha_1 \quad \cdots \quad \Theta; \Gamma \vdash P_n \triangleright \Delta \cdot k : \alpha_n}{\Theta; \Gamma \vdash \sum_{i=1}^n k!m_i.P_i \triangleright \Delta \cdot k : !\sum_{i=1}^n m_i.\alpha_i} \\
\mathcal{T}_{\text{PAR}} : \frac{\Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta'}{\Theta; \Gamma \vdash P \parallel Q \triangleright \Delta \odot \Delta'} \quad (\Delta \bowtie \Delta') \\
\mathcal{T}_{\text{INACT}} : \frac{}{\Theta; \Gamma \vdash 0 \triangleright \Delta} \\
\mathcal{T}_{\text{DEF}} : \frac{\Theta \cdot A : \tilde{t}\tilde{\alpha}; \Gamma \cdot \tilde{x} : \tilde{t} \vdash P \triangleright \Delta \cdot \tilde{k} : \tilde{\alpha} \quad (A(\tilde{x}\tilde{k}) = P)}{\Theta \setminus A; \Gamma, \tilde{y} : \tilde{t} \vdash A(\tilde{y}\tilde{h}) \triangleright \Delta \cdot \tilde{h} : \tilde{\alpha}} \\
\mathcal{T}_{\text{VAR}} : \frac{}{\Theta \cdot A : \tilde{t}\tilde{\alpha}; \Gamma, \tilde{y} : \tilde{t} \vdash A(\tilde{y}\tilde{h}) \triangleright \Delta \cdot \tilde{h} : \tilde{\alpha}}
\end{array}$$

**Fig. 2.** Typing system for the calculus  $\mathcal{L}$ .

for  $x$  is a pair composed by the session type  $\alpha$  and its dual,  $\bar{\alpha}$  being the derived type for the session opened on  $x$ . Rule  $\mathcal{T}_{\text{CTH}}$  types a *catch* action, assigning to the captured session  $k'$  a type  $\alpha$ , which is provided by the rule  $\mathcal{T}_{\text{THR}}$  in the corresponding *throw* action. Rules  $\mathcal{T}_{\text{IN}}$  and  $\mathcal{T}_{\text{OUT}}$  define the expected type for a sum of input and output actions, respectively, where with abuse of notation, we still use  $m_i$  to denote the type of the message  $m_i$ . Rule  $\mathcal{T}_{\text{PAR}}$  defines the synchronization among processes having compatible types; the resulting type is given by their composition. Finally, the rules  $\mathcal{T}_{\text{DEF}}$  and  $\mathcal{T}_{\text{VAR}}$  define the types for process definitions, where the information accumulated on the basis  $\Theta$  about process variables may be used to type recursive definitions. We assume that the range of  $\Delta$  in  $\mathcal{T}_{\text{INACT}}$  and  $\mathcal{T}_{\text{VAR}}$  contains only  $0$  and  $\perp$ .

If a type sequent  $\Theta; \Gamma \vdash A(\tilde{x}\tilde{k}) \triangleright \Delta$  is derivable from the typing system, we say that the process  $A$  is *typable*, and its type, denoted by  $[A]$ , is determined by the types associated to each link  $x$  of  $A$  in  $\Gamma$ , and to each session  $k$  of  $A$  in  $\Delta$ . We write  $[A]_x$  to denote the session type for  $x$  in the process  $A$ . Given a link  $x$  of a typable component  $A$ , we will denote by  $k_x$  the session that  $A$  opens on link  $x$ . Then, we have that  $[A]_x$  is the session type  $\alpha$  (respectively,  $\bar{\alpha}$ ) such that  $x : \langle \alpha, \bar{\alpha} \rangle$  is in  $\Gamma$ , and the session  $k_x$  has type  $\alpha$  (respectively,  $\bar{\alpha}$ ) in  $\Delta$ . Thus, from the point of view of  $A$ , the type of a link  $x$  is the type of the session  $k_x$  opened on that link. Hence, we usually write  $[A]_x$  and  $[A]_{k_x}$  interchangeably.

Let us show now which are the types corresponding to the FTP system of our example. Starting from the daemon component, its type  $[Daemon]$  is given by the type `DAEMON_b`, assigned accordingly to the typing system in Fig. 2 to the session  $b$  that a daemon establishes with the server:

```
DAEMON_b = ? DAEMON2_b
DAEMON2_b = ?(get. !(result.DAEMON_b + nosuchfile.DAEMON_b) + quit.0)
```

On the other hand, the type of the server component  $[Server]$  defines two session types: `SERVER_a` for the session  $a$  it establishes with the client, and `SERVER_b` for the session  $b$  it establishes with the daemons:

```
SERVER_a = ? ?user. ?pass.
          !( rejected. 0 + connected. !(SERVER_b). 0 )
```

and, from rule  $\mathcal{T}_{THR}$ , we have that `SERVER_b` is precisely the type dual of `DAEMON_b` (and notice that by definition of  $\odot$  their composition is  $\perp$ ). Consequently, the type for the whole FTP system, obtained composing the typings of the server and the daemons, is:

$$[System] = \{ a : [Server]_a, b : [Server]_b \odot [Daemon]_b \} = \{ a : SERVER\_a, b : \perp \}.$$

We can see several interesting differences between the specification in  $\mathcal{L}$  of the components previously shown and their corresponding session types above. First of all, session types describe the behaviour of the component during a single session (that is, from the point where a new session name is created in an *accept/request*, to the moment where this session is no more used, and its name is lost). Hence, session types describe usually a finite pattern of actions (as it can be seen in particular in the session types of the server component). Second, session types separate the interleaving of actions performed on different session names, allowing a modular description of interactions. On the contrary, the process algebra specification describes the full behaviour of the components, interleaving actions from different sessions (as in the case of the server component), and is usually recursive to an *accept* or *request* action.

Now, we could write the session type of an FTP client component perfectly compliant (that is, dual) with the server and the daemon described so far, ensuring successful interaction among them. However, our objective is to deal with component adaptation, and for this reason, a more interesting (although very simple) client is represented below (on the left the component, on the right the corresponding type):

```
Client(server) = server!request(c).      CLIENT_c = !
                    c!login(usr,pwd).      !login.
                    c!download(filename).    !download.
                    c?data(filedata). 0      ?data. 0
```

It is easy to see that the above client is not compatible with our FTP system (and obviously the corresponding components would deadlock when composed in parallel). Indeed, the more relevant differences between them are:

- The name of the messages used in both types are different, though we could guess some correspondences between them (e.g., `get` and `download`).

- Also the protocols of the components are different. For instance, the `DAEMON_b` type recurs until a `quit` message is received, while the client just performs a single `download` command.
- The client ignores the throwing of a daemon session from the server. Instead, its behaviour is represented by a single session.
- Finally, the client unwisely ignores any error message coming from the server.

### 3 Adaptor specification

In this section, we introduce a simple, high-level notation for describing the intended correspondence among the functionality of two components being adapted. This correspondence is specified by means of a mapping that establishes a number of rules relating messages of both components. This specification will be then used for the automatic construction of an adaptor that mediates the interaction of the two components.

A detailed description of the notation for adaptor specification can be found in [2]. Here, we will show only how we can use it for accommodating the differences between the types `SERVER_a` and `CLIENT_c` above. The intended adaptation between them may be represented by the following specification:

```

S = { !login(usr,pwd)      <> ?user(usr), ?pass(pwd);    // r1
      <> !rejected(msg);    // r2
      <> !connected();      // r3
      !download(filename) <> ?get(filename);           // r4
      !download(filename) <> ;                          // r5
      ?data(filedata)     <> !result(filedata);        // r6
      ?data("No such file") <> !nosuchfile();          // r7
      ?data("Not connected") <> ;                     // r8
      <> ?quit(); }                                       // r9

```

where the actions of the client are represented in the left terms while those of the server (and its daemons) are on the right.

The specification `S` establishes a correspondence between messages in both components. For instance, the `login` output message in the client is mapped to a pair of `user` and `pass` actions in the server, as indicated in the first rule. Instead, server's messages `rejected`, `connected` or `quit` have no correspondence in the other part (rules `r2`, `r3`, and `r9`). Finally, some other messages, like client's `data` may correspond to different actions in the server, like `result` (rule `r6`), `nosuchfile` (rule `r7`), or even to no action at all (rule `r8`).

An adaptor specification defines the properties that the behaviour of an adaptor component must satisfy [2]. Each rule in a specification can be translated into a term in the process calculus  $\mathcal{L}$ . For instance, for the rule `r1`, we have:

$$R1(1,r) = 1?login(usr,pwd).( r!user(usr).0 \parallel r?pass(pwd).0 \parallel R1(1,r) )$$

meaning that if the adaptor accepts a message `login` from the component at its left (represented by the session `1`), then it will eventually perform one action `user` and one action `pass` in its interaction with the component at its right.

Given an adaptor specification, we say that a process  $A$  satisfies it, if  $A$  fulfills the rules of the specification. Formally, this means that the process  $A$  is simulated by the parallel composition of the rules of the specification [2].

The specification  $S$  above provides a minimal description of an adaptor that plays the role of “component-in-the-middle” between the FTP system and its client, mediating their interaction. The ultimate goal of such a specification is to obtain an adaptor, a component both satisfying the specification, and providing the required adaptation between the components being adapted.

Formally, the notion of adaptor is introduced as follows.

**Definition 1 (Adaptor).** *Let  $\alpha_P$  and  $\beta_Q$  be sessions types for two components  $P$  and  $Q$ , respectively, and let  $S$  be an adaptor specification. A process  $A(l, r)$  is an adaptor for  $\alpha_P$  and  $\alpha_Q$  under  $S$  iff:*

1.  $A(l, r)$  satisfies  $S$ , and
2.  $[A]_l \bowtie \alpha_P$  and  $[A]_r \bowtie \alpha_Q$ .

The adaptor  $A$  is a process with two session types,  $([A]_l$  and  $[A]_r)$  —one for each component to be adapted—, compatible with the corresponding session types  $\alpha_P$  and  $\alpha_Q$ . The two conditions a process has to satisfy to be an adaptor ensure that: (i) the process follows the adaptation pattern given by the adaptor specification, and (ii) the parallel composition of the adaptor with the components  $P$  and  $Q$  is “safe”, as we will illustrate later.

Given an adaptor specification, and the session types of the components to be adapted, an automatic procedure [2] deploys a concrete adaptor (not a type, but an actual component, here represented by a process in  $\mathcal{L}$ ), that both satisfies the specification, and adapts the mismatching behaviour of the actual components represented by those session types. For instance, for the previous specification  $S$ , and the session types of the FTP client and server (`CLIENT_c` and `SERVER_a`, respectively), a possible result of the generation procedure is the adaptor:

```
Adaptor(l,r) =
  l?accept(x). r!request(y). x?login(usr,pwd). y!user(usr). y!pass(pwd).
  ( y?connected(). y?(z). x?download(filename). z!get(filename).
    ( z?result(filedata). x!data(filedata). z!quit(). 0
      + z?nosuchfile(). x!data("No such file"). z!quit(). 0 )
    + y?rejected(msg). x?download(filename). x!data("Not connected"). 0 )
```

The adaptor above has two link names  $l$  and  $r$  for interacting with its left and right counterparts (the client and the server, respectively). In the first two actions, a session is opened on each of these links, and then the adaptor begins to interact with the client and the server following the behaviour represented in their session types, and according to the correspondence between messages indicated in the specification  $S$ . Thus, the initial client’s message `login` is transmitted to the server by means of one `user` and one `pass` message (rule `r1` in  $S$ ). If the server replies with a `connected` message (without correspondence in the client, as per rule `r3`), then a daemon session  $z$  will be accepted by the adaptor and the interaction goes on through this session (note how the client remains

completely unaware of this fact). The following client's action `download` will be transmitted by a `get` (rule `r4`), and the reply of the daemon (either `result` or `nosuchfile`) will be returned to the client with a `data` message (rules `r6` and `r7`). Anyway, in both branches the client closes its session at this point, and the adaptor also ends by sending the `quit` message (rule `r9`) the server is waiting for. On the contrary, if the server rejects the connection and closes the session, client's message `download` will not be transmitted (rule `r5`). Instead, the client will be replied with a "Not connected" indication in a `data` message.

### 3.1 Safe composition

As we already mentioned, the conditions to be fulfilled by an adaptor guarantee the safe interaction of the components to be adapted. In order to clarify what we mean by that, we introduce the notion of *session-safety*.

**Definition 2 (Session safety).** *A process  $P$  in  $\mathcal{L}$  is session safe for a set of links  $L$  if for every trace  $P \xrightarrow{*} E \not\rightarrow$ , we have that either:*

1.  $E \equiv 0$ , or
2. if  $E \xrightarrow{\xi}$  then  $\xi = x?acp(k)$  or  $\xi = x!rqt(k)$  for some link  $x \in L$  and some session  $k$ .

Session safety states that a process does not deadlock in the middle of the computation of a session. In other words, once a session is open, then it will finish without deadlocks. We now prove that the definition of adaptor ensures the conditions for guaranteeing that the interactions are safe.

**Proposition 1.** *Let  $P, Q, A$  be three processes sharing only two sessions  $(k_l, k_r)$ , such that  $[P]_{k_l} \bowtie [A]_{k_l}$ ,  $[Q]_{k_r} \bowtie [A]_{k_r}$ , and for every session  $k \notin \{k_l, k_r\}$  we have  $[P]_k = [A]_k = [Q]_k = \perp$ . Assume that  $fn(P) \cap fn(Q) = \emptyset$  (i.e.  $P$  and  $Q$  do not interact each other). Then, for  $E = P \parallel A \parallel Q$  we have that either:*

1.  $E \equiv 0$ , or
2. if  $E \not\rightarrow$ , and  $P \xrightarrow{\xi}$  or  $Q \xrightarrow{\xi}$ , then for some link  $x$ ,  $\xi \in \{x!rqt(k_l), x?acp(k_l), x!rqt(k_r), x?acp(k_r)\}$ , or
3.  $E \longrightarrow P' \parallel A' \parallel Q'$  where:
  - (a)  $P' = P$ ,  $[A']_{k_l} = [A]_{k_l}$ ,  $[A']_{k_r} \bowtie [Q']_{k_r}$ , or
  - (b)  $Q' = Q$ ,  $[A']_{k_r} = [A]_{k_r}$ ,  $[A']_{k_l} \bowtie [P']_{k_l}$

*Proof.* Let us suppose  $E \neq 0$ ,  $E \not\rightarrow$ , and  $P \xrightarrow{\xi}$  (analogously if  $Q \xrightarrow{\xi}$ ). If, for any link  $x$ ,  $\xi \notin \{x!rqt(k_l), x?acp(k_l)\}$ , then  $\xi$  is one of the following transition labels:  $k_l!thw(k)$ ,  $k_l?cth(k)$ ,  $k_l!m$ , or  $k_l?m$  (for some session  $k$  and message  $m$ ). In any of these cases, as  $[P]_{k_l} \bowtie [A]_{k_l}$ , the process  $A$  would present the corresponding complementary action, and therefore  $P \parallel A \longrightarrow$ , which contradicts the original hypothesis ( $E \not\rightarrow$ ). If  $E \longrightarrow P' \parallel A' \parallel Q'$ , then (as  $P$  and  $Q$  does not share names) the interaction comes from a synchronization made by  $A$  and either  $P$  or  $Q$ . Let us assume the interaction is produced by the parallel composition of  $A$  and  $P$ . Then, clearly  $Q' = Q$ . Moreover, since the only

session shared by  $A$  and  $P$  is  $k_l$ , we will get  $[A']_{k_r} = [A]_{k_r}$ . Finally, it is easy to prove (by structural induction on the transition system of  $\mathcal{L}$ ) that  $[A']_{k_l} \bowtie [P']_{k_l}$ . Analogously, if the interaction is performed between  $A$  and  $Q$ .  $\square$

Proposition 1 establishes a correspondence between a property on session types (compatibility) and the way the corresponding processes proceed. The theorem below generalizes this result, and proves that a deadlock-freedom result for the parallel composition of two processes can be deduced from the compatibility of the corresponding session types. Of course, this result is only derivable when processes do not interact (among them or with others) through other sessions.

**Theorem 1.** *Let  $P, Q, A$  be three processes sharing only two links  $\{l, r\}$  such that  $fn(P) \cap fn(Q) = \emptyset$ , and for every session  $k \notin \{k_l, k_r\}$  we have  $[P]_k = [A]_k = [Q]_k = \perp$ . If  $A$  is an adaptor for  $[P]_l$  and  $[Q]_r$  under a certain specification  $S$ , then  $P \parallel A \parallel Q$  is session safe for  $\{l, r\}$ .*

*Proof.* We will prove the following equivalent result. One of the following situations hold, for every trace  $P \parallel A \parallel Q \longrightarrow^* E$ :

- i.  $E \equiv 0$ , or
- ii. if  $E \not\rightarrow$  and  $E \xrightarrow{\xi}$ , then  $\xi \in \{l!rqt(k_l), l?acp(k_l), r!rqt(k_r), r?acp(k_r)\}$ , or
- iii.  $E = P' \parallel A' \parallel Q'$  where  $[A']_{k_l} \bowtie [P']_{k_l}$  and  $[A']_{k_r} \bowtie [Q']_{k_r}$ .

Now, we reason by induction on the trace length. The base case is trivial, because  $E = P \parallel A \parallel Q$ , and then we can apply previous Proposition. Let us assume the statement is true for every trace with length  $n$  (inductive hypothesis). Then, if we consider a  $(n+1)$ -trace  $P \parallel A \parallel Q \longrightarrow^n E' \longrightarrow E$ , by applying the inductive hypothesis to the first  $n$  transitions, we obtain  $E' = P' \parallel A' \parallel Q'$  where  $[A']_l \bowtie [P']_l$  and  $[A']_r \bowtie [Q']_r$  (note that two first statements of the inductive hypothesis are not applicable because  $E' \longrightarrow E$ ). Therefore, by Proposition 1,  $E = P'' \parallel A'' \parallel Q''$  satisfying either (3.a) or (3.b). In both cases,  $P'', A''$  and  $Q''$  satisfy the desired requirements.  $\square$

Notice that the conditions of the theorem ensure that the components being adapted will not deadlock in their interaction with other possible components of the system, and this is the sense of enforcing that the type of every session on links different from  $l$  or  $r$  is  $\perp$ . If this were not the case, obviously we could not ensure session-safety since a deadlock occurred in another session would deadlock the whole component, including sessions  $k_l$  and  $k_r$ .

Returning to our example, the types corresponding to the `Adaptor(1,r)` component above are:

```
ADAPTOR_l = ? ?login. ?download. !data. 0
ADAPTOR_r = ! !user. !pass. (?connected. ?(ADAPTOR_z). 0 + ?rejected. 0)
ADAPTOR_z = !get. ( ?result. !quit. 0 + ?nosuchfile. !quit. 0 )
```

and it can be easily proved both that the adaptor satisfies the specification  $S$  and also that its types are compatible with those of the client and the server: `CLIENT_c`  $\bowtie$  `ADAPTOR_l` and `ADAPTOR_r`  $\bowtie$  `SERVER_a`. Hence, we are under the conditions of Theorem 1, and we can conclude that the system:

$$Client(l) \parallel Adaptor(l, r) \parallel Server(r, s) \parallel \prod_{i=1}^n Daemon(s)$$

is session-safe for the links  $\{l, r\}$ , and now these components are able to interact successfully. Indeed, after the client performs its FTP session, both the client and the adaptor end, and the FTP server and its daemons are again in their original states, all of them expecting to perform an accept action, requested by a new client (that may need a completely different adaptation).

## 4 Concluding remarks

The objective of this paper was to set a formal foundation for the adaptation of heterogeneous components that may present mismatching interaction behaviour. The three main ingredients of the methodology of software adaptation that we began to develop in [2] can be synthetised as follows:

- IDL component interfaces are extended with an explicit description of the interaction behaviour of a component;
- the desired adaptation is specified via a simple set of (possibly nondeterministic and partial) correspondences between actions of two components;
- a concrete adaptor is automatically generated (if possible), given its partial specification and the interfaces of two components.

In this paper we improved the methodology with the adoption of *session types* to describe the interaction behaviour of components. As shown in Sect. 2, session types feature a disciplined, modular representation of behaviour at a suitable level of abstraction. Most importantly, their type discipline permits a rigorous type checking of component protocols that paves the way for rigorous analysis.

The adoption of session types also permits to describe recursive component protocols, which could not be fully achieved in [2] where component interfaces could only declare finite, non-recursive interaction patterns followed by a component. It is important to stress that while session types permit to describe recursive protocols, their analysis and verification remains tractable.

One of the main contributions of this paper is the formal definition of the notion of *adaptor*, and the proof that any adaptor for (the session types of) two components guarantees their correct session-wise interaction (Sect. 3).

Session types were originally introduced in [6]. Our treatment of session types however differs from [6] in that we employ a more expressive process algebra (and formally define its operational semantics), while simplifying the sometimes cumbersome notation used for session types in [6]. Moreover, our notion of type compatibility relies on the notion of subtyping introduced in [12], rather than coinciding with the more restrictive notion of type duality used in [6].

One of the advantages of using session types is that they can cope with heterogeneous descriptions of component interfaces. Namely, if the protocols of different components have been expressed using different formalisms, such

protocols can be typed into the corresponding session types, providing so an homogeneous way of dealing with the composition of third-part components.

Indeed, we argue that the introduction of behavioural types in component interfaces is necessary to achieve a systematic component-based software development, capable of guaranteeing properties of the resulting systems. The results presented in this paper are a step in this direction. Our plans for future work include the integration of the proposed adaptation methodology with available CBSE environments so as to promote its experimentation and assessment.

Space limitations do not allow a proper comparison of our adaptation methodology with other proposals, including those centering on the introduction of connectors in software architectures (e.g., [8]). A thorough comparison of our adaptation methodology with other proposals can be found in [2], while a detailed comparison between session types and other formalisms is reported in [6].

## References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, 1997.
2. A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software, Special Issue on Automated Component-Based Software Engineering*, 2003. (in press). A preliminary version of this paper was published in *Component deployment*, LNCS 2370, pages 185–199. Springer, 2002.
3. A.W. Brown and K.C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–47, 1998.
4. G.H. Campbell. Adaptable components. In *ICSE 1999*, pages 685–686. IEEE Press, 1999.
5. G.T. Heineman. An evaluation of component adaptation techniques. In *ICSE'99 Workshop on CBSE*, 1999.
6. K. Honda, V.T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *European Symposium on Programming (ESOP'98)*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
7. P. Inverardi and M. Tivoli. Automatic synthesis of deadlock free connectors for COM/DCOM applications. In *ESEC/FSE'2001*. ACM Press, 2001.
8. A. Lopes, M. Wermelinger, and J.L. Fiadeiro. Higher-order connectors. *ACM Transactions on Software Engineering and Methodology*, 12(1):64–104, 2003.
9. J. Magee, S. Eisenbach, and J. Kramer. Modeling Darwin in the  $\pi$ -calculus. In *Theory and Practice in Distributed Systems*, LNCS 938, pages 133–152. 1995.
10. B. Morel and P. Alexander. Automating component adaptation for reuse. In *Proc. IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 142–151. IEEE Computer Society Press, 2003.
11. A. Vallecillo, J. Hernández, and J.M. Troya. New issues in object interoperability. In *Object-Oriented Technology*, LNCS 1964, pages 256–269. Springer, 2000.
12. A. Vallecillo, V.T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. *ENTCS*, 68(3), 2003.
13. D.M. Yellin and R.E. Strom. Protocol specifications and components adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.