Component adaptation through flexible subservicing [1]

Antonio Brogi

Department of Computer Science, University of Pisa, Italy
Email: brogi@di.unipi.it

Carlos Canal Ernesto Pimentel

Department of Computer Science, University of Málaga, Spain
Email: canal@lcc.uma.es, ernesto@lcc.uma.es

Abstract

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering (CBSE). We present here a formal methodology for the soft adaptation of components presenting mismatching interaction behaviour. The notions of access rights (associating components with the services they are allowed to use) and subservicing (providing alternative services in place of those requested by components lacking the required access rights) are exploited to feature a secure and flexible adaptation of third-party components.

1 Introduction

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering [8,10,14]. The possibility for application builders to adapt off-the-shelf software components for working properly within their applications is a must for the development of a true component marketplace, and for component deployment in general [7]. Available component-oriented platforms address software interoperability at the signature level, typically by means of Interface Description Languages (IDLs). IDLs

[1] 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).

are a sort of *lingua franca* for specifying the functionalities offered by heterogeneous components that were developed in different languages. IDL interfaces defining the signature of the methods offered by a component are an important step towards software integration, since they solve signature mismatches in the perspective of adapting or wrapping components to overcome such differences. It is also at this signature level where mismatching between data formats is usually solved, for instance by means of XML descriptions. However, even if all signature problems may be overcome, there is no guarantee that the components will suitably interoperate, as mismatches may also occur because of the differences in the protocols defining the behaviour of the components [16]. While case-based testing can be performed to check the compatibility of software components, more rigorous techniques are needed to lift component integration from hand-crafting to an engineering activity.

In our previous work [4], we have developed a formal methodology for the adaptation of components presenting mismatching interaction behaviour. The main ingredients of the methodology can be summarised as follows:

(1) *Component interfaces.* IDL interfaces are extended with a formal description of the behaviour of the components, which explicitly declares the interaction protocols they follow.
(2) *Adaptor specification.* Adaptor specifications are simply expressed by sets of correspondences among messages of the components. The distinguishing aspect of the used notation is that it results in a high-level, partial specification of the adaptor.
(3) *Adaptor derivation.* Given its partial specification and the interfaces of two components, a concrete adaptor is automatically generated. The separation of adaptor specification and derivation permits the automation of the error-prone and time-consuming task of implementing a detailed adaptor, simplifying the task of the software developer.

A limitation of the adaptation technique described in [4] is that it is somewhat rigid, in that it only succeeds if there exists an adaptor that strictly satisfies the given specification. Indeed, in many situations an adaptor could be nevertheless deployed by weakening some of the requirements stated in the specification.

In this paper, we extend the methodology presented in [4] precisely to overcome this type of limitation. The notions of access rights (associating components with the services they are allowed to use), and subservicing (providing alternative services in place of those requested by components lacking the required access rights), are exploited to feature a secure, soft adaptation of third-party components. From a technical viewpoint, we rely on *session types*, firstly defined in [11] to describe non-terminating behaviour by means of true types.

It is worth outlining that while we use the notion of access rights to enforce a secure adaptation of component services, we do not deal here with other important aspects of security, such as authentication, which must be additionally employed to enforce a secure communication between the components. Neither we deal with incompatibility of data representations in different components, which are being addressed by current component technologies [7].

The rest of the paper is organised as follows. In Sect. 2 we introduce a simplified example of a Video-on-Demand service, which will be used throughout the paper to illustrate the methodology. Session types are introduced in Sect. 3, after defining a process calculus to denote component protocols, and their applicability to the example is illustrated in Sect. 4. Sect. 5 describes the application of the methodology of adaptor specification and derivation to allow the successful interoperation of components presenting mismatching interaction behaviour. Finally some concluding remarks are drawn in Sect. 6.

2 A Video-on-Demand Service

We shall exemplify the use of the methodology in terms of a (simplified) Video-on-Demand (VoD) system. The VoD is a Web service providing access to a database of movies and news.

There are four different profiles of clients. Each profile grants certain access rights. On the one hand, registered users —those paying a regular fee— are divided into *news*, *movies*, and *full* clients. On the other hand, *guests* are unregistered (possibly occasional) users.

Guests are only allowed to *search* for a movie in the VoD catalog, *preview* it for a few minutes, and *quit* the system. Clients with profile *news* have the same capabilities as guests, but they may also watch the news. The *movies* profile grants access to *view* movies but not the news, while *full* clients may access both news and movies. Once a certain movie has been selected for viewing it, normal (*movies*) users may *start* its visualization, while *full* users may also decide to *record* it permanently in their computers.

If a client tries to *view* a movie without having the rights for that, the system will treat such request as a *preview* request. Similarly, invalid attempts to *record* a movie will be interpreted as *start* requests.

When a client opens a session with the VoD system, it follows a connection procedure which associates the session with one of the four profiles described, depending on the identity of the client, and generates an adaptor for connecting to the system. The connection consists of the following steps:

- The client asks the VoD system for its behavioural interface definition, which describes the services provided.
- After analysing this interface and comparing it with its own interface (which may use different command names and interaction protocols), the client makes a connection proposal, in the form of an adaptor specification between the two interfaces.
- The client requests to open a session to the VoD system, identifying itself via an authentication procedure (not shown here), and sends the adaptor specification to the VoD, together with its own behavioural interface.
- The system, given the access rights of the client, the adaptor specification, and both the interfaces of the client and its own, constructs an adaptor component.
- If during the construction the adaptor specification cannot be fully satisfied, the system also returns a modified adaptor specification that must be accepted by the client before using it for the interaction with the system.

3 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 [3,12,13]. 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 types* introduced in [11].

Session types present some important features that distinguish them from processes written in a general process algebra like the π-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, these restrictions make session types a calculus much more tractable than other studied alternatives, like the

π-calculus or CSP. The advantages of employing session types instead of other concurrency formalisms are thoroughly discussed in [17].

Under this approach, a program is considered as a collection of sessions, each one being a chain of diadic interactions. Each session is designated by a private channel, through which interactions belonging to that session are performed. The use of diadic sessions for the specification of software interaction allows a modular specification of complex systems. The objective 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.

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.

3.1 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 [11]. Apart from some simplification in the notation, the main difference is that we allow the alternative composition of output actions (somehow equivalent to an *if-then-else* construction), and not only of input actions as in [11]. We give also a transition system for the calculus, not present in [11].

The base sets are: link names, ranging over $\mathcal{N}$; session names, ranging over $\mathcal{S}$; data values, ranging over $\mathcal{D}$; and process names, ranging over $\mathcal{P}$. Then, 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_i denotes a process, A is a process identifier, x is a link name, k and k' are 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 agent 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 agent identifier, even A itself.

We consider two kinds of actions in the process calculus $\mathcal{L}$: output actions

$$\text{REQ}: \frac{}{x!request(k).P \xrightarrow{x!rqt(k)} P} \quad \text{THR}: \frac{}{k!throw(k').P \xrightarrow{k!thw(k')} P}$$

$$\text{ACC}: \frac{}{x?accept(k).P \xrightarrow{x?acp(h)} P\{h/k\}} \ (h \notin fn(P) \setminus \{k\})$$

$$\text{CTH}: \frac{}{k?catch(k').P \xrightarrow{k?cth(h)} P\{h/k'\}} \ (h \notin fn(P) \setminus \{k'\})$$

$$\text{SUM}: \frac{}{\sum_{i=1}^{n} \lambda_i.P_i \xrightarrow{\lambda_j} P_j} \ (j = 1 \cdots n)$$

$$\text{PAR}_{open}: \frac{P \xrightarrow{x!rqt(k)} P' \quad Q \xrightarrow{x?acp(k)} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

$$\text{PAR}_{throw}: \frac{P \xrightarrow{k!thw(k')} P' \quad Q \xrightarrow{k?cth(k')} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

$$\text{SYNC}: \frac{P \xrightarrow{k!m} P' \quad Q \xrightarrow{k?m} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \quad \text{PAR}: \frac{P \xrightarrow{\lambda} P'}{P \parallel Q \xrightarrow{\lambda} P' \parallel Q}$$

$$\text{IDE}: \frac{P\{\tilde{y}/\tilde{x}, \tilde{h}/\tilde{k}\} \xrightarrow{\lambda} P'}{A(\tilde{y}\tilde{h}) \xrightarrow{\lambda} P'} \ (A(\tilde{x}\tilde{k}) = P)$$

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

($k!m_i$), where a message selector m_i is sent through a session k, and input actions ($k?m_i$), where a message selector m_i is received from a session k. There are four special message selectors: *request*, *accept*, *throw*, and *catch*. All of them include 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 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.

The labelled transition relation described in Fig. 1 defines the operational semantics of $\mathcal{L}$. The first four rules describe the behaviour of prefix actions concerning session manipulations. Both (ACC) and (CTH) rules model session capturing. Thus, both *accept* and *catch* actions receive a new session name

not occurring in $fn(P)$ (free names of P). Rule (SUM) defines the behaviour of a sum of (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. To model the parallel composition of processes we have four different transition rules. (PAR) describes the behaviour of a parallel composition where one of the processes presents a labelled transition. The other three rules define different forms of synchronization: (PAR$_{open}$) and (PAR$_{throw}$) model session opening and session throwing, respectively, whereas (SYNC) models the synchronous exchange of input and output messages. Additionally to the transition system in Fig. 1, we assume also standard commutativity and associativity axioms for choice (+) and parallel ($\|$) composition operators.

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.

3.2 Typing system

Whereas the type system defined in [11] 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 type system could be easily extended to deal also with data. We will denote by *TExp* the set of type expressions constructed by following grammar:

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

where α, α_i, β are type expressions, and Λ is a type identifier, where for each identifier Λ exists a unique defining equation $\Lambda = \alpha$. The constant type 0 represents inaction's type, and $\bot$ denotes a specific type indicating that no further connection is possible at a given link. In other words, if a link k has a type $\bot$ in a process definition, then k is not offered by this process as an open link. Type expressions $!\alpha$ and $?\alpha$ correspond to *request* and *accept* primitives, respectively, whereas $!(\alpha)$ and $?(\alpha)$ correspond to *throw* and *catch* primitives. The expression t_i denotes the type associated to a message (which will coincide with the message selector). Then, the type $? \sum_i t_i.\alpha_i$ denotes the type of the branching behaviour, given by a process which is waiting with several options, and which behaves as type α_i if the i-th action is selected. Similarly, $! \sum_i t_i.\alpha_i$ represents the complementary behaviour, with respect to output actions.

Now, we define a *typing* as a partial mapping Δ from session names to types in *TExp*. 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.

Given a type α where $\perp$ does not occur, we define its co-type $\bar{\alpha}$ (also called its dual type), as follows:

$$\begin{array}{llll} \overline{?\alpha} = !\overline{\alpha} & \overline{?(\alpha).\beta} = !(\alpha).\overline{\beta} & \overline{!\sum_i t_i.\alpha_i} = ?\sum_i t_i.\overline{\alpha_i} & \overline{0} = 0 \\ \overline{!\alpha} = ?\overline{\alpha} & \overline{!(\alpha).\beta} = ?(\alpha).\overline{\beta} & \overline{?\sum_i t_i.\alpha_i} = !\sum_i t_i.\overline{\alpha_i} & \end{array}$$

The co-type of a given type denotes the complementary behaviour of the original type, and $\overline{\overline{\alpha}} = \alpha$. It is obvious that the composition of a type and its co-type is successful, in the sense that the corresponding components will execute without deadlock [11]. However, imposing a condition of duality seems too restrictive in the context of real software components.

In [17] a notion of type compatibility is defined in terms of a subtyping relation and type duality. Intuitively speaking, a session type α is a subtype of β if α can be used in any context where β is used, and no error occurs in the session. Basically, this means that α should have more —or equal— branchings (input alternatives), and less —or equal— selections (output alternatives). Based on this subtyping relation, we say that α is compatible with β, denoted by $\alpha \bowtie \beta$, if α is a subtype of the dual of β. This notion of compatibility can be naturally extended to typings. Defined this way, compatibility ensures successful composition of the corresponding components, too. (More details about these subtyping and compatibility relationships can be found in [17].)

When two typings, Δ_1 and Δ_2, are compatible, their composition $(\Delta_1 \odot \Delta_2)$ is defined as a new typing given by:

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

The typing system for the calculus $\mathcal{L}$ is shown in Figure 2, and deals with sequents of the form:

$$\Theta; \Gamma \vdash P \triangleright \Delta$$

which mean: "under the current environment, given by Θ and Γ, the process P has a typing Δ". As in [11], Θ denotes a *basis*, which is a mapping from process variables (i.e. process names) to the corresponding arguments (links and sessions) types, and the *sorting* $(t \in)\Gamma$ stores types for links, which are expressed by means of sorts like $\langle \alpha, \bar{\alpha} \rangle$. 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*. Given a typing

$$\mathcal{T}_{\text{ACC}} : \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta; \Gamma, x : \langle \alpha, \bar{\alpha} \rangle \vdash x?accept(k).P \triangleright \Delta \cdot k : ?\alpha}$$

$$\mathcal{T}_{\text{REQ}} : \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \bar{\alpha}}{\Theta; \Gamma, x : \langle \alpha, \bar{\alpha} \rangle \vdash x!request(k).P \triangleright \Delta \cdot k : !\alpha}$$

$$\mathcal{T}_{\text{IN}} : \frac{\Theta; \Gamma \vdash P_1 \triangleright \Delta \cdot k : \alpha_1 \quad \cdots \quad \Theta; \Gamma \vdash P_n \triangleright \Delta \cdot k : \alpha_n}{\Theta; \Gamma \vdash \sum_{i=1}^{n} k?m_i.P_i \triangleright \Delta \cdot k : ?\sum_{i=1}^{n} m_i.\alpha_i}$$

$$\mathcal{T}_{\text{OUT}} : \frac{\Theta; \Gamma \vdash P_1 \triangleright \Delta \cdot k : \alpha_1 \quad \cdots \quad \Theta; \Gamma \vdash P_n \triangleright \Delta \cdot k : \alpha_n}{\Theta; \Gamma \vdash \sum_{i=1}^{n} k!m_i.P_i \triangleright \Delta \cdot k : !\sum_{i=1}^{n} m_i.\alpha_i}$$

$$\mathcal{T}_{\text{THR}} : \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \beta}{\Theta; \Gamma \vdash k!throw(k').P \triangleright \Delta \cdot k : !(\alpha).\beta \cdot k' : \alpha}$$

$$\mathcal{T}_{\text{CTH}} : \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \beta \cdot k' : \alpha}{\Theta; \Gamma \vdash k?catch(k').P \triangleright \Delta \cdot k : ?(\alpha).\beta}$$

$$\mathcal{T}_{\text{PAR}} : \frac{\Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta'}{\Theta; \Gamma \vdash P \parallel Q \triangleright \Delta \odot \Delta'} \; (\Delta \bowtie \Delta')$$

$$\mathcal{T}_{\text{INACT}} : \frac{}{\Theta; \Gamma \vdash 0 \triangleright \Delta}$$

$$\mathcal{T}_{\text{DEF}} : \frac{\Theta \cdot A : \tilde{t}\tilde{\alpha}; \Gamma \cdot \tilde{x} : \tilde{t} \vdash P \triangleright \Delta \cdot \tilde{k} : \tilde{\alpha}}{\Theta \setminus A; \Gamma, \tilde{y} : \tilde{t} \vdash A(\tilde{y}\tilde{h}) \triangleright \Delta \cdot \tilde{h} : \tilde{\alpha}} \; (A(\tilde{x}\tilde{k}) = P)$$

$$\mathcal{T}_{\text{VAR}} : \frac{}{\Theta \cdot A : \tilde{t}\tilde{\alpha}; \Gamma, \tilde{y} : \tilde{t} \vdash A(\tilde{y}\tilde{h}) \triangleright \Delta \cdot \tilde{h} : \tilde{\alpha}}$$

Fig. 2. A simplified typing system for the calculus $\mathcal{L}$.

(or sorting or basis) Ξ, we write $\Xi \cdot k : \alpha$ to denote the mapping $\Xi \bigcup \{k : \alpha\}$ provided that $k \notin dom(\Xi)$.

As we have already mentioned, the typing system in Figure 2 is similar to that provided in [11], but adapted to the process calculus $\mathcal{L}$. The two first rules define the sort associated to a link x, on which an *accept* or *request* is made. Notice that the sort for x is a pair composed by the session type α and its dual, α being the derived type for the session opened on x. Rules $\mathcal{T}_{\text{IN}}$ and $\mathcal{T}_{\text{OUT}}$ define the expected type for a sum of input and output actions, respectively. Rules $\mathcal{T}_{\text{THR}}$ and $\mathcal{T}_{\text{CTH}}$ describe how to type *throw* and *catch* actions, respectively. $\mathcal{T}_{\text{PAR}}$ defines the synchronization among processes having

compatible types; the resulting type is given by their composition. Finally, the rules $\mathcal{T}_{\mathrm{DEF}}$ and $\mathcal{T}_{\mathrm{VAR}}$ define the types for process definitions, where the information accumulated on the basis Θ about process variables may be used to type recursive definitions. We assume that the range of Δ in $\mathcal{T}_{\mathrm{INACT}}$ and $\mathcal{T}_{\mathrm{VAR}}$ contains only 0 and $\perp$.

If a type sequent $\Theta; \Gamma \vdash A(\tilde{x}) \triangleright \Delta$ is derivable from the typing system, we say that the process $A(\tilde{x})$ is *typable*, and its type Δ is denoted by $[A(\tilde{x})]$. When the process arguments can be deduced from the context, we use $[A]$ for short.

The type of a process $A(\tilde{x})$ will be given by the session types associated to the sessions opened on each link name x_i in $\tilde{x}$. We write $[A]_{x_i}$ to denote the session type for x_i 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 α (respectively, $\bar{\alpha}$) such that $x : \langle \alpha, \bar{\alpha} \rangle$ is in Γ, and the session k_x has type α (respectively, $\bar{\alpha}$) in Δ. 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.

4 Service specification

We now present a specification of the VoD service described in Sect. 2 both in terms the process calculus $\mathcal{L}$ and of session types. However, component specifications will be normally provided by session types, which can be derived from their process implementations.

4.1 Behaviour of the VoD service: Process description

We will consider that the `VoD` service is connected to its clients using a link `a`, on which client requests for `vod` sessions are accepted. For each request, a session `daemon` is opened with a `VoDDaemon` to which the `VoD` system is connected by a link `b`. This daemon will be in charge of handling the interaction with the client. Hence, the `daemon` session is handed over (`throw`) to the client, and the `VoD` returns to its initial state, allowing concurrent access to the system.

```
VoD(a,b) = a?accept(vod). b!request(daemon).
              vod!throw(daemon). VoD(a,b)
```

Once the `VoDDaemon` accepts the session opened by the `VoD` system, it is ready to input different commands from the client. Each command implies a certain sequence of messages to be exchanged (i.e., a protocol). For instance, after

selecting a movie with the `view` command, the client may either issue a `start` or a `record` command to start visualization or recording. Finally, the session `vod` ends when the client quits, and the `VoDDaemon` is ready to handle a new client session.

```
VoDDaemon(b) = b?accept(daemon). VoDSession(b,daemon)

VoDSession(b,daemon) = daemon?search(title).
                          daemon!list(movies). VoDSession(b,daemon)
                     + daemon?preview(item). VoDStream(b,daemon)
                     + daemon?view(item).
                          ( daemon?start(). VoDStream(b,daemon)
                          + daemon?record(). VoDStream(b,daemon) )
                     + daemon?news(date). VoDStream(b,daemon)
                     + daemon?quit(). VoDDaemon(b)
```

The transmission of video data is performed by `VoDStream` via an output action `stream` which must be acknowledged by `ok` for indicating a correct reception of the data, or by `retry` for indicating the need of re-transmission (for instance, because of network errors).

```
VoDStream(b,daemon) = daemon!stream(video).
                        ( daemon?ok(). VoDSession(b,daemon)
                        + daemon?retry(). VoDStream(b,daemon) )
```

4.2 Behaviour of the VoD service: Type description

As we mentioned previously, the processes `VoD` and `VoDDaemon` may be considered as an implementation of the behaviour of a component, expressed in the process calculus $\mathcal{L}$. In this section we will show the typings Δ_{VoD} and $\Delta_{VoDDaemon}$, respectively corresponding to these processes. These typings define the session types describing the interaction between the VoD service and its clients through links `vod` and `daemon` —that is, the session types $[\,VoD]_{vod}$ and $[\,VoDDaemon]_{daemon}$. For short, we will simply call them `VOD` and `DAEMON`, respectively.

These session types will be used as the specification of the VoD service. According to rules in Figure 2, the type of each session can be automatically derived from the corresponding process. Thus, the VoD service is specified by the following session types:

```
VOD   = ?
        !(DAEMON). 0

DAEMON = ?( search(TITLE). !list(STRING). DAEMON
```

```
          + preview(ITEM). STREAM
          + view(ITEM).
               ?( start(). STREAM  +  record(). STREAM )
          + news(DATE). STREAM
          + quit(). 0 )

STREAM = !stream(VIDEO).
            ?( ok(). DAEMON
             + retry(). STREAM )
```

The session type `VOD` refers to the initial session established between the client and the VoD service. Notice how session types allow a modular description of component behaviour. Notice also that the interactions between the `VoD` process and the `VoDDaemon` are not shown in `VOD`, since they correspond to a different session (the one using link `b`).

Session type `VOD` just indicates that a session type `DAEMON` is thrown to the client. After that, the session type ends (though the process `VoD` does not), and all the interactions with the client will be held directly by `DAEMON`.

On the other hand, the session type `DAEMON` refers to the actual client session, representing the actions exchanged between the client and the VoD daemon. Again, the session ends when the client quits (though the `VoDDaemon` process is ready to open a new session).

Notice that the behaviour of the service is described completely independent of other important system issues like access rights and subservices, thus following the principle of separation of concerns typical of aspect-oriented software development (AOSD). The motivation here is to use access rights and subservices as a way of configuring the system: As it will be shown, the VoD service will behave differently (by means of adaptation) depending on user access rights and subservice availability.

4.3 Access rights and subservice definition

The session types of the VoD service describe the potential behaviour of the system when a session is opened by a client. However, this information is not enough to connect safely client components to the VoD.

In particular, session types do not include information on the access rights that correspond to the available services. For instance, as discussed in Sect. 2, a *view* action should be available only to clients with a *movies* or *full* profile. This information must be provided by the component as part of its interface description. For this reason, we complement the protocol description with a

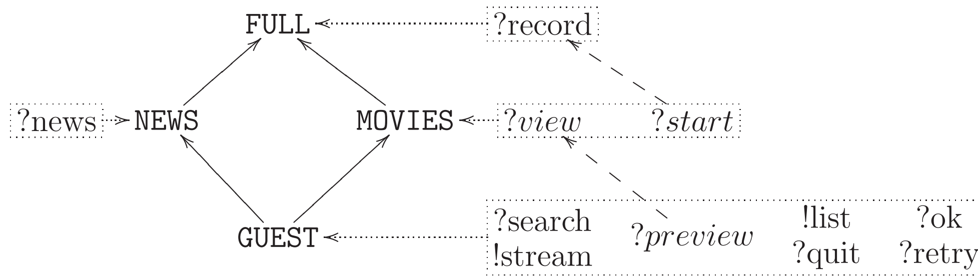

Fig. 3. Client profiles (solid arrows), access rights (dotted arrows), and subservices (dashed arrows) in the VoD system.

hierarchy of client profiles, and a description of the access rights that correspond to each profile, as shown in left-hand side of Figure 3. Obviously, actions that are accessible to a given profile are also accessible to those higher in the graph.

Typically, if a client opens a session and requests a service without owning the appropriate access rights, the system will reply by raising an exception without providing the service. However, in many cases this may be considered too strict, and a more flexible behaviour would be desirable. For instance, the system could provide a *subservice* —a different service, accessible by the client— as an alternative to the one requested. Intuitively speaking, a subservice is a kind of surrogate of a service which features only a limited functionality of such service.

Hence, we extend component interfaces so as to include information about subservices. It is worth observing that our approach allows to feature such flexible adaptations without having to modify or complicate protocol specifications. Subservices are specified by defining a partial order on actions, which can be depicted as a graph, as in the right-hand side of Fig. 3. For instance, the action `preview` is considered as a subservice of `view`. So, when a *guest* client sends a request for viewing a movie, the system will answer by offering only a preview of it. Similarly, `start` is considered a subservice of `record`.

5 Adaptor specification and generation

We now introduce a simple, high-level notation for describing the intended *mapping* among the functionalities of two components being adapted. This description will be used for the automatic construction of an *adaptor* that mediates their interaction.

Let us consider the specification of a possible client of our VoD system, as represented by the session type `CLIENT` below:

```
CLIENT = !
         !menu(). ?info(MOVIELIST).
            !( play(TITLE). ?data(VIDEO). 0
             + download(TITLE). ?data(VIDEO). 0 )
```

Initially the `CLIENT` requests a session to the `VOD` system (which is represented by the initial action `!`). Then, it asks for the list of movies (`!menu`) available in the VoD database, and decides either to `!play` or to `!download` one of them. In either case, it will expect a video stream by means of an input action `?data`.

The mismatch between the specification of the `VOD` system and that of the `CLIENT` is both *syntactic* and *behavioural*. Syntactic mismatch deals with discrepancies in message names and/or parameters in both components (e.g., `?data` vs. `?stream`), while behavioural mismatch deals with protocols and command ordering (for instance, the client assumes to talk directly to the VoD system, while the latter will use a specific daemon for managing each client session, or the fact that the confirmation protocol for video transmission is ignored in the client).

Syntactic mismatch will be solved by specifying a mapping describing the intended connection between both components. Then, behavioural mismatching will be solved (if possible) by a process that builds the adaptor, as shown in Sect. 5.2.

5.1 Adaptor specification

As for syntactic discrepancies, we observe that while there may exist one-to-one correspondences between some commands in both components, adaptation does not simply amount to matching or translating message names. Indeed, more general relations (one-to-many, one-to-none, many-to-many) may occur even in a simple example like this. Moreover, we may also find mismatching parameters between corresponding commands in either part.

For this reason, we are interested in adapting not trivial mismatches where, for instance, reordering and remembering of messages is required. The adaptor will be specified by means of a *mapping* that associates actions and data of two components. For instance, the mapping expressing the adaptation required for our example consists of the following association rules:

```
M = { !menu()           <> ?search("");               // 1st
      ?info(string)     <> !list(string);             // 2nd
      !play(title)      <> ?view(title), !start();    // 3rd
      !download(title)  <> ?view(title), !record();   // 4th
```

```
    ?data(video)      <> !stream(video), ok?();     // 5th
                      <> ?quit()                    // 6th
  }
```

where, as a convention, all the actions on the left hand side of each rule refer to the first of the components being adapted (in this case the `CLIENT`), while those on the right refer to the second one (here, the `VOD` system).

Each rule in a mapping can be (automatically) translated into a process algebra term that defines a certain property that the behaviour of any adaptor component must satisfy. For instance, the semantics of the first rule in `M` is given by the term `m1` below:

```
m1(l,r) = l?menu(). ( r!search(""). 0 || m1(l,r) )
```

which indicates that whenever the component on the left (here the `Client`, represented by the session `l`) performs a `!menu` command, the adaptor must ensure that the component on the right (here the `VOD`, represented by the session `r`) will eventually —though not necessarily in that particular moment— perform a `?search` action with an empty string in place of the missing title.

A full description of the formal semantics of mappings is out of the scope of this paper, though it can be found in [6]. Here, let us simply say that for a given mapping `M`, we will denote by Π(`M`) the parallel composition of the properties defined by the rules in `M`. The set of *possible* adaptors defined by a mapping `M` is then the set of processes which are *simulated* by the process Π(`M`) (notice that in general Π(`M`) denotes an infinite set of processes). As we will show in Section 5.2, the adaptor generation process not only takes into account the mapping `M`, but also the session types representing the behaviour of the components being adapted, in order to explore this set Π(`M`) for finding an *actual* adaptor for these components.

The second rule in `M` maps `?info` to `!list`. Notice here how the real parameter `string` makes explicit the correspondence between data in the actions mapped, instanciating both the formal parameters `STRING` and `MOVIELIST` declared in the session types of the components. Real parameter names have a global scope in the mapping, so that every occurrence of a certain parameter name, even if in different rules, refers to the same parameter.

In the third rule the use of one-to-many correspondences between actions is shown. A single action in the client (`!play`) is mapped to two different actions in the VoD system (`?view` *and* `?start`). The same occurs in the fourth rule, now with the action `!download`. Moreover, these two rules establish a non-deterministic correspondence between VoD's input action `?view` (which appears in both of them), and *either* client's `!play` or `!download` output

actions.

The fifth rule contains again a one-to-many mapping between actions. It states that whenever the system issues a `!stream` action, a `?data` action will be performed by the client (transmitting this way the `video` data), but also that a confirming `?ok` action will be received by the VoD. When generating the adaptor we will show how this rule is used to solve the mismatch in the protocol of video transmission between the VoD system and the client, which neither confirms nor asks to retry the transmission.

The sixth rule in `M` indicates that VoD's action `quit` has no correspondence in the client, so that it may be matched by the adaptor whenever the VoD system requires it.

Finally, the mapping also states implicitly (by not referring to them) that the rest of the services in the VoD (`news`, `preview`, etc.) are not required for this client, and that they will not be used in the adaptation process.

Hence, the mapping `M` provides the minimal specification of an adaptor that will play the role of "component-in-the-middle", mediating the interaction between the VoD and the client. It is worth noticing that the adaptor specification defined by a mapping abstracts from many details of component behaviour. The burden of dealing with these details is left to the automatic process of adaptor construction, described in Sect. 5.2.

5.2 Adaptor generation

In the previous section, we have shown how the intended connection between two software components —the `VoD` system and a certain `Client`— is specified by means of a mapping `M`. Given such a specification `M`, and the session types `VOD` and `CLIENT`, respectively describing the `VoD` and the `Client` components, a concrete adaptor component `A` (if any) will be generated by means of a fully automated procedure. The adaptor will fulfill the syntactic matching between `VoD` and `Client` as stated in the mapping, and it will also solve all behavioural mismatches between the actual protocols followed by the two components.

Notice that the adaptor is a process (i.e., a real component) and not a type. However, it will be derived directly from the session types `CLIENT` and `VOD`, and not from the component implementation represented by the processes `Client` and `VoD`.

We now sketch the implementation of the algorithm that constructs a soft adaptor. Roughly speaking, given two session types describing the behaviour of the components to be adapted, and a mapping specifying the intended

start of revision

adaptation between them, the adaptor returned will be a *component-in-the-middle* such that:

(1) The parallel composition of the adaptor with the session types of the two components will not deadlock, and
(2) the adaptor will satisfy the action correspondences and data dependecies specified by the mapping, by possibly introducing some sub-servicing according to the access rights of the components involved.

Because of its inherent nondeterministic nature, the algorithm has been implemented in Prolog. Because of space limitations, we focus on the main skeleton of the algorithm, and omit some obvious predicate definitions in favour of a shorter description of their usage. (A detailed description of the basic algorithm for adaptor generation can be found in [4].)

The top-level predicate is `adapt/4` which, given the interfaces of two components and an adaptor specification, returns an adaptor that allows the two components to interoperate according to the given adaptor specification. Component interfaces are represented by terms of the form `interface(Tp,Dp)`, where `Tp` represents the session type exposed by a component and `Dp` its access rights and subservice declarations. Predicate `transform_spec/4` expands the given specification `Spec` with new correspondence rules that are obtained by replacing services with sub-services in the rules of `Spec` in all possible ways. The construction of the adaptor is then obtained in a generate-and-test fashion, by first constructing (predicate `find_adaptor/5`) a candidate adaptor for the components and then verifying (predicate `satisfy/2`) that such process satisfies the (expanded) adaptor specification.

```
adapt(interface(Tp,Dp),interface(Tq,Dq),Spec,Adaptor) :-
   transform_spec(Dp,Dq,Spec,NewSpec),
   find_adaptor(NewSpec,par(Tp,Tq),nil,nil,Adaptor),
   satisfy(Adaptor,NewSpec).
```

Predicate `find_adaptor/5`, which basically implements a loop which tries to incrementally build an adaptor by progressively eliminating all the deadlocks that may occur in the parallel composition of the adaptor with the session types of the two components. Predicate `find_adaptor(Spec,Context,Adaptor,Last,NewAdaptor)` inputs an adaptor specification (`Spec`), the session types of two components (`Context`), a partially constructed `Adaptor`, and the last action (`Last`) added to the adaptor, and it returns an adaptor (`NewAdaptor`) if there exists one, otherise it fails.

```
find_adaptor(_,Context,Adaptor,_,Adaptor) :-
   deadlocks(and(Context,Adaptor),[]).

find_adaptor(Spec,Context,Adaptor,Last,NewAdaptor) :-
```

```
    deadlocks(and(Context,Adaptor),[_|_]).
    states_after_last(and(Context,Adaptor),Last,States),
    deadlocks(States,[D|Ds]),
    successes(States,[]),
    choose_unlocking_action(Spec,[D|Ds],A),
    add(A,Last,Adaptor,Adaptor1),
    find_adaptor(Spec,Context,Adaptor1,A,NewAdaptor).

find_adaptor(Spec,Context,Adaptor,Last,NewAdaptor) :-
    deadlocks(and(Context,Adaptor),[_|_]).
    states_after_last(and(Context,Adaptor),Last,States),
    deadlocks(States,[]),
    prefix_of_last(Last,Adaptor,NewLast),
    find_adaptor(Spec,Context,Adaptor,NewLast,NewAdaptor).
```

The first rule specifies that if the parallel composition of the adaptor with the session types of the two components cannot deadlock, then the given adaptor is returned as result and the constructions halts as there are no more deadlocks to unlock. The absence of deadlocks is checked by predicate `deadlocks/2` which returns the list of possible deadlocked states.

When the parallel composition of the adaptor with the session types of the two components may instead deadlock, then either the second or the third rule applies — depending on whether the evolutions allowed by the last action added to the adaptor may lead or not to deadlocks. This check is implemented by predicate `states_after_last`, which collects in its third argument the set of states reachable by executing action `Last`, and by predicate `deadlocks/2`.

The second rule models the case in which the evolutions allowed by the last action added to the adaptor may lead to deadlocks. Predicate `successes/2` is used to verify that no success states can be reached after executing `Last` (i.e., a success state is a state from which no deadlock can be reached). Indeed if both deadlocked and success states can be reached after executing `Last`, then the algorithm has to remove `Last` from the adaptor and backtrack — as any attempt to add a new action after `Last` will unavoidably spoil those success states. If instead only deadlocked states (and no success state) can be reached after executing `Last`, then an action `A` capable of unlocking one of these deadlocks is chosen nondeterministically and used to expand (predicate `add/4`) the adaptor as one of the possible actions following `Last`. The construction process continues then with a tail recursion, where `A` becomes the new last action added to the adaptor. Predicate `choose_unlocking_action/3` is in charge of nondeterministically choosing an action `A` capable of unlocking one of the deadlocked states. Notice that such a choice is constrained by the action correspondences and data dependencies established by the adaptor specification, and may lead to introducing subservicing when a component

does not have the needed access rights to perform a required action.

The third rule applies when the evolutions allowed by the last action added to the adaptor do not lead to deadlocks. In that case, there is no point in trying to expand further the adaptor "below" `Last`. The algorithm hence backtracks one step behind, by considering the action `NewLast` that prefixes `Last` in the `Adaptor` to continue recursively.

The following description of the example should be revised according to the new algorithm, in particular by (1) introducing the expanded mapping, and (2) adjusting the name and parameters of the predicates.

revision ends here

In order to illustrate how the algorithm works, let us consider again our example. For a given client of the VoD service, several adaptors could be developed, according to the client access rights and subservice definitions.

When the client requests a session with the `VoD` system, the session is assigned a particular access right, which will be used for constructing the corresponding adaptor. Suppose that `movies` is the access right corresponding to the `CLIENT` component described in Section 4.2. The algorithm is executed with the predicate call:

```
?- find_adaptor(vod_st,client_st,m,movies,A).
```

where `vod_st` and `client_st` represent the session types `VOD` and `CLIENT`, `m` represents the mapping `M`, and `A` is the variable which will be instantiated to the adaptor constructed.

The adaptor will communicate with the components `Client` and `VoD` by means of two links, that we may call `client` and `vod`, respectively. Initially, the adaptor `A` is the empty process `0` and the parallel composition of the adaptor, the client, and the VoD service (represented by their corresponding session types) presents two deadlocks: the client is deadlocked in a request action on link `client` (as indicated by the initial `!` in the session type `CLIENT`), while the VoD service is deadlocked in an accept action on link `vod` (as indicated by the initial `?` in `VOD`). Let us suppose that the construction starts by expanding `A` with an action triggering the first of these deadlocks:

```
A = client.accept?(c).
```

and `last` points to this initial action. Since there are still deadlocks after `last`, construction goes on expanding the adaptor with a new action that now triggers the deadlock transition `accept` in the VoD service:

```
A = client.accept?(c). vod.request!(v).
```

Hence, the we have established two sessions —that we have called `c` and `v`—, the first one with the client and the second with the VoD, matching so the actions `!` and `?` in the session types of the components.

Now, the adaptor can be expanded with two different actions, each triggering one deadlock. These actions are `v?(d)`, catching a session `d` to match `VOD`'s action `!(DAEMON)`, on the one hand, or `c?menu()`, matching `CLIENT`'s action `!menu`. Once again, the adaptor will be eventually expanded with both these two actions, the order not being relevant:

```
A = client.accept?(c). vod.request!(v). c?menu(). v?(d).
```

At this point, the session `VOD` ends, but the process of adaptation goes on with the session `DAEMON` catched by the adaptor with `v?(d)`. Notice that the client remains unaware of this change of sessions, and it will go on communicating through session `c`.

The types show now that the components are deadlocked on actions `?info` of the client part, and also in `?search`, `?preview`, `?view`, etc. of the VoD. Of all these input actions the adaptor only knows —from the first rule of the mapping— how to fill the parameter of action `?search`, so this is the action selected, and the adaptor expands into:

```
A = ... c?menu(). v?(d). d!search("").
```

The only action that can be matched now is VoD's action `!list`. When doing this, and according to the second rule in `M`, the adaptor gets the data parameter `string` to match client's `?info`. Hence, the adaptor becomes (two steps in one):

```
A = ... v?(d). d!search(""). d?list(string). c!info(string).
```

The `CLIENT` session is now deadlocked on two alternative actions `!play` and `!download`. Let us suppose that the first one is chosen first. Similarly to what we have seen till now, the adaptor will be expanded with `c?play(title)`, and after that, in several subsequent steps to:

```
A = ... c!info(string). c?play(title). d!view(title). d!start().
        d?stream(video). d!ok(). c!data(video). d!quit(). 0
```

by applying the rules corresponding to these actions in the mapping `M`. Since VoD action `!quit` is mapped to `NONE`, the adaptor may match freely this action when required by the `VOD`.

At this point, both the types `CLIENT` and `DAEMON` end. Thus, there are not more deadlocks after the last action added, and this branch of the adaptor construction is finished. However, there are still deadlocks in the interaction

tree of the components, so the construction backtracks to the point of the alternative between `!play` and `!download`. Now, the second action is matched, resulting in the adaptor:

```
A = ... c!info(string). ( c?play(title). ... d!quit(). 0
                        + c?download(title). d!view(title).
```

However, when the adaptor tries to match VoD's action `!record` to fulfill the fourth rule in the map, it notices that the client profile (`movies`) does not allow to access this service. Hence, its subservice `!start` is matched instead:

```
A = ... + c?download(title). d!view(title). d!start().
```

and the adaptor construction goes on as before, rendering at the end the full adaptor component:

```
A = client.accept?(c). vod.request!(v).
      c?menu(). v?(d). d!search(""). c!info(string).
      ( c?play(title). d!view(title). d!start().
          d?stream(video). d!ok(). c!data(video). d!quit(). 0
      + c?download(title). d!view(title). d!start().
          d?stream(video). c!data(video). d!ok(). d!quit(). 0 )
```

which will allow the client and VoD components to interact without deadlocks, satisfying the access rights and subservice definitions, and fulfilling the mapping `M` suggested by the client (except for the command `?record` for which the client did not have rights, and for which the subservice `?start` was offered instead).

Hence, the algorithm will return the adaptor `A` together with a new version of the mapping (where the rules modified are marked with asterisks) due to access rights and subservices definition in the VoD system:

```
{ !menu()            <> ?search("");              // 1st
  ?info(string)      <> !list(string);            // 2nd
  !play(title)       <> ?view(title), !start();   // 3rd
  !download(title)   <> ?view(title), ?start();   // 4th ***
  ?data(video)       <> !stream(video), ok?();    // 5th
                     <> ?quit()                   // 6th
}
```

so that the client may decide whether the proposed adaptor component still satisfies its needs.

As a final example, let us suppose that the profile of the client component is `guest`, which would not allow it to view movies. The construction of the

adaptor would be more or less the same as before until we arrive to the point in which client actions `!play` and `!download` are to be matched. According to the mapping given by the client, these actions should be matched to `view` in the VoD system. However, the client has no rights to use this service, so the subservice `preview` will be matched instead:

```
A’ = ... c?play(title). d!preview(title).
```

Since the profile `guest` does not allow either `start` or `record`, the adaptor construction would proceed to:

```
A’ = ... c?play(title). d!preview(title). d?stream(video).
         d!quit . c!data(video). 0
```

and the algorithm will return the adaptor `A’` together with the modified mapping:

```
{ !menu()            <> ?search("");              // 1st
  ?info(string)      <> !list(string);            // 2nd
  !play(title)       <> ?preview(title);          // 3rd ***
  !download(title)   <> ?preview(title);          // 4th ***
  ?data(video)       <> !stream(video), ok?();    // 5th
                     <> ?quit()                   // 6th
}
```

6 Concluding remarks

In this paper we have tried to illustrate the main aspects of a formal methodology for the development of adaptors capable of solving behavioural mismatches between heterogeneous interacting components. The proposed methodology extends the adaptation technique described in [4] by featuring a *soft* adaptation of software components when the given adaptation requirements cannot be fully satisfied. Technically this is achieved by exploiting the notion of *subservice* to suitably weaken the initial specification when needed. Correspondingly, component interfaces are extended with a declaration of their subservice relations as well as with the *access rights* needed to access the component services. The separation between component protocols, access rights, and subservice declarations, follows the separation of concerns advocated by aspect-oriented development, and supports the flexible configuration of existing components in view of their (dynamic) adaptation.

Our work falls in the well-established research stream which advocates the application of formal methods to describe the interactive behaviour of software systems. A thorough comparison of our adaptation methodology with other

proposals is discussed in [4]. A distinguishing feature of our approach consists of adopting *session types*, firstly defined in [11], to describe (possibly non-terminating) component behaviour by means of true types. The adoption of session types reduces sensibly the complexity of verifying properties w.r.t. other approaches based on full-fledged process algebras, while featuring an expressiveness bonus w.r.t. the approaches based on finite state machines [11].

It is also interesting to relate our notion of subservice to the concept of action refinement in process algebras [2,9]. In short, action refinement establishes a relation between actions —more precisely, between one atomic action `a` and one process `A`—, by which each occurrence of the action `a` in a process `P` is refined or replaced by the full process `A`, resulting in a process `P'` which may be considered as a transformation of `P` at a lower level of abstraction (i.e. a refinement). Our notion of subservicing also establishes a kind of substitution between actions. However, while action refinements define fixed substitutions, that must be performed always, subservices express more general and adaptative transformations by which an action is replaced by its substitutes (since they may be more than one action) only when it is necessary for finding an adaptor.

The issue of assembling off-the-shelf components using *wrappers* to encapsulate components and to enforce security policies has been recently addressed in [15], where an extension of the π-calculus is proposed to express compositions. However, the focus of [15] is different from ours. While our goal is the automatic deployment of adaptors, [15] illustrates how security properties of given wrappers can be formulated and verified —although in an *ad-hoc* (non-systematic) way.

In the example used throughout the paper, we have always supposed our components to be complete. For instance, our client just needed an adaptor for its connection to the VoD system. However, an interesting question, deserving future research, would be how to assemble such a client component from different parts, where each part may match one or more services in the VoD system.

Finally, while we used the notion of access rights to enforce a secure adaptation of component services, we did not deal with other important aspects of security, such as authentication or secrecy protocols (e.g., [1]), which will have to be additionally employed to enforce a secure communication among the components. An interesting question for our future work is how to deal with access rights that may change dynamically.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of ACM*, 46(5):749–786, 1999.

[2] L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. *Information and Computation*, 115(2):179–247, December 1994.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, 1997.

[4] 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*, 2004. (in press). A preliminary version of this paper was published in *Component Deployment*, LNCS 2370, pages 185–199. Springer, 2002.

[5] A. Brogi, E. Pimentel, and A. Roldán. Compatibility of Linda-based component interfaces. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 66(4), 2002.

[6] A. Brogi, C. Canal, and E. Pimentel. On the specification of software adaptation. In *Proceedings of FOCLASA'03*, *Electronic Notes in Theoretical Computer Science (ENTCS)* (in press), 2003.

[7] A.W. Brown and K.C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–47, 1998.

[8] G.H. Campbell. Adaptable components. In *ICSE 1999*, pages 685–686. IEEE Press, 1999.

[9] Roberto Gorrieri and Arend Rensink. Action refinement. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2000.

[10] G.T. Heineman. An evaluation of component adaptation techniques. In *ICSE'99 Workshop on CBSE*, 1999.

[11] 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.

[12] P. Inverardi and M. Tivoli. Automatic synthesis of deadlock free connectors for COM/DCOM applications. In *ESEC/FSE'2001*. ACM Press, 2001.

[13] J. Magee, S. Eisenbach, and J. Kramer. Modeling Darwin in the π-calculus. In *Theory and Practice in Distributed Systems*, LNCS 938, pages 133–152. 1995.

[14] B. Morel and P. Alexander. Automating Component Adaptation for Reuse. In *IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 142–151, IEEE Computer Society Press, 2003.

[15] P. Sewell and J. Vitek. Composition of untrusted code: Wrappers and causality types. *Journal of Computer Security*, 2003.

[16] A. Vallecillo, J. Hernández, and J.M. Troya. New issues in object interoperability. In *Object-Oriented Technology*, LNCS 1964, pages 256–269. Springer, 2000.

[17] A. Vallecillo, V.T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 68(3), 2003.