Component adaptation through flexible subservicing

Antonio Brogi\textsuperscript{a}, Carlos Canal\textsuperscript{b,∗}, Ernesto Pimentel\textsuperscript{b}

\textsuperscript{a} Department of Computer Science, University of Pisa, Italy
\textsuperscript{b} Department of Computer Science, University of Málaga, Spain

Received 15 May 2004; received in revised form 15 April 2005; accepted 15 July 2005
Available online 12 July 2006

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.

Keywords: Component-based software development; Software adaptation; π-calculus; Session types

1. Introduction

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering (CBSE) [9,13,20]. The possibility for application builders to adapt off-the-shelf software components to work properly within their applications is a must for the development of a true component marketplace, and for component deployment in general [8]. Available component-oriented platforms address software interoperability at the signature level, typically by means of Interface Description Languages (IDLs). IDLs 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 [25,7]. 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.

∗ 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).

* Corresponding author. Tel.: +34 95 213 33 11; fax: +34 95 213 13 97.
E-mail addresses: brogi@di.unipi.it (A. Brogi), canal@lcc.uma.es (C. Canal), ernesto@lcc.uma.es (E. Pimentel).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
In our previous works [4,5], 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, rather than on a process algebra like the \( \pi \)-calculus as in our previous work. Session types are defined in [23,15,16] to describe non-terminating behaviour by means of types instead of processes. In fact, if we view concurrent computation as a collection of interaction structures, types of computing naturally arise as a mathematical encapsulation of those structures [15]. Furthermore, a type discipline for processes yields an effective typing procedure, providing a way of checking the consistency of communication patterns between interacting parties. Specifically, session types ensure the lack of run-time errors including inconsistency between communication patterns, which is an important discipline in parallel, communication based programming [23].

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 do we deal with incompatibility of data representations in different components, which are being addressed by current component technologies [8].

The rest of the paper is organised as follows. In Section 2 we introduce an example of a Video-on-Demand service, which will be used throughout the paper to illustrate the methodology. Session types are introduced in Section 3, after defining a process calculus to denote component protocols, and their applicability to the example is illustrated in Section 4. Section 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 Section 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, 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 [3,17,18]. 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 have an important drawback, that is 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 [16].

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.** Instead of representing the data values interchanged between components by means of input/output operations, session types refer to the corresponding data types. Indeed, session types are a type discipline for concurrent programs, reflecting the situation in type-checking procedural languages, where signatures indicate the types of the formal parameters of procedures, abstracting from the real parameter values passed in actual procedure calls. In fact, data values are closely related to component implementation (e.g. to the internal decision taking mechanisms that depend on specific values of data), and it seems reasonable to abstract from them in component interfaces. As a consequence of this abstraction, the space of states for analysis is greatly reduced, without much loss of expressiveness.

- **Sessions are limited to diadic communications between two components.** Although meaningful examples of multi-party interactions can be produced, most of these cooperations can be expressed by means of several separate diadic interactions, where the only information missing is some causal relations between actions (e.g. a certain component sends a message to a second one as a result of having received a previous message from a third one, a causal relation which in most cases is irrelevant from the point of view of this second component). Once again, the effect is that the state of spaces and the complexity of analysis are reduced, since diadic interactions will be much simpler than the corresponding multi-parties obtained by interleaving them. However, although sessions are diadic it will be shown that general $n$-ary links can be shared by any number of components (e.g. a server with a bundle of clients, each one performing a diadic session with the server).

- **Mobility is explicitly represented.** Instead of using standard input/output actions for sending both data or channels, like in the $\pi$-calculus, mobility is explicitly represented in session types by using special throw/catch actions. The result is not only a discipline in specification that makes clearer the description of dynamic communication topologies, but also a clean type abstraction for the resulting operational structures [15]. Notice that a side effect of this characteristic is that since sessions are diadic, once a component throws a session, it cannot use it anymore.

- **Mixed alternatives are not allowed.** Following many proposals in the field (see for instance [21]), input and output actions cannot be combined in a single non-deterministic choice. Consequently there are two different alternative operators: one for input and one for output. In fact, this agrees with normal design practice (e.g. in object-oriented
systems), in which at a given point an object or component either (a) is waiting for some external event to occur (typically, the invocation of one of its methods), and this will be represented by an input alternative; or (b) is performing some computation (and then, possibly deciding on sending one of several messages to other objects), which will be represented by an output alternative; but not both.

It is worth noting that these restrictions are not relevant limitations in our context, as we will show throughout this paper. On the contrary, these restrictions make session types a calculus much more tractable than other studied alternatives — e.g. process algebras such as the π-calculus or CSP. Indeed, process algebras are Turing-complete languages, whose expressive power causes them to be more about implementation than specification of useful and interesting properties [19]. In spite of their formal semantics, that would allow using them for model-checking-style verification, usually analysis is not feasible in practice, running out of time or space and having no idea whether a given process operates according to the model. Of necessity, the expressive power of these type systems is below Turing-completeness. More detailed discussions on the advantages of employing session types instead of other concurrency formalisms can be found in [23,15,16,26].

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 presented by Honda et al. in [16]. 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 [16]. We give also a transition system for the calculus, not present in that paper.

The syntax of the process calculus $\mathcal{L}$ is defined as follows:

\[
P ::= 0 \mid \text{act}\,P \mid \sum_i s!m_i.P_i \mid \sum_i s?m_i.P_i \mid P_1 \parallel P_2 \mid A(\tilde{s})
\]

\[
\text{act} ::= l\text{?request}(s) \mid l\text{?accept}(s) \mid \text{s!throw}(s') \mid s\text{?catch}(s')
\]

where 0 represents the empty process, $P_i$, $P_1$, $P_2$, $P_i$ denote processes, $A$ is a process identifier, $l$ denotes a link name, $s$, $s'$ denote session names, $\tilde{s}$ denotes a sequence of names, and $m_i$ denotes a message, syntactically composed by a message name and a sequence of data arguments.

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

We consider two kinds of action in the process calculus $\mathcal{L}$: output actions ($s!m_i$), where a message $m_i$ is sent through a session $s$, and input actions ($s?m_i$), where a message $m_i$ is received from a session $s$. There are four special message names: 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 $l$ waits for the acceptance (accept) of a session on this link. When these two actions synchronize, a fresh session name 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 us to change the topology of the system dynamically.

As is seen, $\mathcal{L}$-processes are, in essence, sequential processes running in parallel, communicating via persistent link names, so the processes look like π-calculus terms. However, the distinct treatment between link and session names are not present in this calculus. In fact, while in $\mathcal{L}$ sessions are locally created by accept|request actions between two processes, these primitives do not exist as such in process calculi, but are mixed with the general name passing notion if any. In contrast, the language $\mathcal{L}$ makes a clear distinction between them, imposing a discipline in programming. Indeed, it should be noticed that while the initiation of one session takes place through a link name, which may be
shared by more than two parties, each action composing a session is done only though the associated private session name. This separation of the notion of session names from link names, suggested by two distinct ways of using names in name passing process calculi such as the π-calculus, turns out to be essential to both a flexible way of composing interactions, and a clean type abstraction for resulting operational structures.

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 \( P \), including all names excepting those used as arguments of accept or catch actions). Rule (SUM) defines the behaviour of a sum of (input or output) actions \( \lambda_j = s ? m_j \) or \( \lambda_j = s ! m_j \), respectively, which is modelled by the usual non-deterministic choice. To model the parallel composition of processes we have four different transition rules. Rule (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: (PARopen) and (PARthrow) 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 the parallel (\( \| \) ) composition operator.

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

Type systems in programming languages allow us to type programs according to their applicative behaviour, and help us in programming in a disciplined fashion, assuring that no type errors can occur during the execution of the program. In the same way, here we need a type abstraction for processes based on their interactive behaviour [23].
Whereas the type system defined in [16] deals both with data and session types, without loss of generality we shall omit data arguments in process defining equations and message arguments. This simplification is not relevant, and our type system could be easily extended to deal also with data.

In fact, considering data parameters in messages in the typing system in Fig. 2 would stand only for allowing signature mismatch checks between components. However, our goal is to deal with behavioural mismatch, and this kind of signature interoperability is not central to our proposal, being currently solved by means of standard practice (e.g. CORBA, XML). Hence, for simplicity we have omitted these details in the typing system in Fig. 2.

We will denote by $TExp$ the set of type expressions constructed by following grammar:

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

where $\alpha$, $\alpha'$, $\alpha_i$ are type expressions, and $\Lambda$ is a type identifier, where for each identifier $\Lambda$ there exists a unique defining equation $\Lambda = \alpha$. The constant type $0$ represents the 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 session $s$ has a type $\bot$ in a process defining equation, then $s$ is not offered by this process as an open link.

Type expressions $!\alpha$ and $?\alpha$ correspond to request and accept primitives, respectively, whereas $!(\alpha).\alpha'$ and $?(\alpha).\alpha'$ correspond to the type of a process that throws (respectively, catches) a session with type $\alpha$ and then behaves as type $\alpha'$. The expression $t_i$ denotes the type associated to a message (which will coincide with the message name). Then, the type $\sum t_i.\alpha_i$ denotes the type of the branching behaviour, given by a process which is waiting with several options, and which behaves as type $\alpha_i$ if the $i$-th action is selected. Similarly, $\sum t_i.\alpha_i'$ represents the complementary behaviour, with respect to output actions.

Now, we define a typing as a partial mapping $\Delta$ 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 $\alpha$ where $\bot$ does not occur, we define its co-type $\bar{\alpha}$ (also called its dual type) as follows:

\[
\begin{align*}
?\alpha &= !\bar{\alpha} \\
!(\alpha).\alpha' &= !(\alpha).\bar{\alpha'} \\
? \sum t_i.\alpha_i &= ? \sum t_i.\bar{\alpha_i} \\
? \sum t_i.\alpha_i' &= ! \sum t_i.\bar{\alpha_i}'.
\end{align*}
\]

The co-type of a given type denotes the complementary behaviour of the original type, and $\bar{\bar{\alpha}} = \alpha$. Since dual types represent mirror behaviour of the corresponding processes, we can expect that the parallel composition of processes with dual types will execute without deadlock, and from that we can arrive at a definition of successful composition of types, such as that in [16]. However, the notion of duality above seems to be a condition far too restrictive in the context of real software components. For instance, we cannot expect that a client requests all the possible services of a server in order to consider their composition successful, or safe. Instead, we should only require the compatibility for the behaviour they have in common. With this intention, in [26] a notion of type compatibility is defined:

Given two types, $\alpha$ and $\alpha'$ we say that they are compatible, denoted by $\alpha \Rightarrow \alpha'$, if $\alpha$ is a subtype of a dual of $\alpha'$.

Compatibility relies on the relation of subtyping for session types defined in [10]. Intuitively speaking, a session type $\alpha$ is a subtype of another session $\alpha'$ if $\alpha$ can be used in any context where $\alpha'$ is used, and no interaction error occurs in the context. Roughly, this means that $\alpha$ should have more or equal branchings (input alternatives), and fewer or equal selections (output alternatives). We assume that subtyping is closed with respect to commutativity of the alternative operator.

This notion of compatibility can be naturally extended to typings. Two typings are considered compatible if the common session types are compatible, i.e., $\Delta_1 \Rightarrow \Delta_2$ iff $\Delta_1(s) \Rightarrow \Delta_2(s)$ for all $s \in dom(\Delta_1) \cap dom(\Delta_2)$. When two typings, $\Delta_1$ and $\Delta_2$, are compatible, their composition $(\Delta_1 \odot \Delta_2)$ is defined as a new typing given by:

\[
(\Delta_1 \odot \Delta_2)(s) = \begin{cases} 
\bot & \text{if } s \in dom(\Delta_1) \cap dom(\Delta_2) \\
\Delta_1(s) & \text{if } s \in dom(\Delta_1) \setminus dom(\Delta_2) \\
\Delta_2(s) & \text{if } s \in dom(\Delta_2) \setminus dom(\Delta_1).
\end{cases}
\]

Defined this way, compatibility ensures successful composition of the corresponding components, too. More details about these subtyping and compatibility relationships can be found in [10,26].
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 mean: “under the current environment, given by $\Theta$ and $\Gamma$, the process $P$ has a typing $\Delta$”. As in [16], $\Theta$ denotes a basis, which is a mapping from process identifiers 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 (or sorting or basis) $\Xi$, we write $\Xi \cdot s : \alpha$ to denote the mapping $\Xi \cup \{s : \alpha\}$ provided that $s \notin \text{dom}(\Xi)$.

As we have already mentioned, the typing system in Fig. 2 is similar to that provided in [16], but adapted to the process calculus $\mathcal{L}$ presented in Section 3.1. The first two rules define the sort associated to a link $l$, on which an accept or request is made. Notice that the sort for $l$ is a pair composed by the session type $\alpha$ and its dual, $\bar{\alpha}$ being the derived type for the session opened on $l$. Rules $T_{\text{IN}}$ and $T_{\text{OUT}}$ define the expected type for a sum of input and output actions, respectively. Rules $T_{\text{THR}}$ and $T_{\text{CTH}}$ describe how to type throw and catch actions, respectively. $T_{\text{PAR}}$ defines the synchronization among processes having compatible types; the resulting type is given by their composition. Finally, the rules $T_{\text{DEF}}$ and $T_{\text{VAR}}$ define the types for process defining equations, where the information accumulated on the basis $\Theta$ about process identifiers may be used to type recursive definitions. We assume that the range of $\Delta$ in $T_{\text{INACT}}$ and $T_{\text{VAR}}$ contains only 0 and $\bot$. It is worth observing that this introduces some non-determinism in the type derivation, because these two rules do not totally characterize the typing $\Delta$. Thus, the type derivation would need a resource.
splitting strategy to divide the session types stored in $\Delta$ when a parallel composition is typed. Some of these issues are discussed in [24], and [14] provides a general view about the inherent complexity of linear proof theory systems. However, these implementation details about linear typing systems are beyond the scope of this paper.

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

The type of a process $A(\tilde{l})$ will be given by the session types associated to the sessions opened on each link name $l_i$ in $\tilde{l}$. We write $[A]_{l_i}$ to denote the session type for $l_i$ in the process $A$. Given a link $l$ of a typable component $A$, we will denote by $s_l$ the session that $A$ opens on link $l$. Then, we have that $[A]_l$ is the session type $\alpha$ (respectively, $\bar{\alpha}$) such that $l : (\alpha, \bar{\alpha})$ is in $\Gamma$, and the session $s_l$ has type $\alpha$ (respectively, $\bar{\alpha}$) in $\Delta$. Thus, from the point of view of $A$, the type of a link $l$ is the type of the session $s_l$ opened on that link. Hence, we usually write $[A]_l$ and $[A]_{s_l}$ interchangeably.

4. Service specification

We now present a specification of the VoD service described in Section 2 both in terms of 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 for retransmission (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 $\Delta_{VoD}$ 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 \([\text{VoD}]_{\text{vod}}\) and \([\text{VoDDaemon}]_{\text{daemon}}\). 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 Fig. 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:

\[
\begin{align*}
\text{VOD} &= \ ? ! (\text{DAEMON}) . \ 0 \\
\text{DAEMON} &= \ ? ( \text{search} . \ ! \text{list} . \ \text{DAEMON} \\
&\quad + \ \text{preview} . \ \text{STREAM} \\
&\quad + \ \text{view} . \ ? ( \ \text{start} . \ \text{STREAM} + \ \text{record}. \ \text{STREAM} ) \\
&\quad + \ \text{news} . \ \text{STREAM} \\
&\quad + \ \text{quit} . \ 0 ) \\
\text{STREAM} &= \ ! \text{stream} . \ ? ( \ \text{ok} . \ \text{DAEMON} + \ \text{retry} . \ \text{STREAM} )
\end{align*}
\]

Notice that according to the typing system in Fig. 2, message parameters are omitted when typing. However, they could be considered without problems, then allowing to check mismatch in the arity of messages or in the types of the parameters.

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 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 client components to the VoD safely.

In particular, session types do not include information on the access rights that correspond to the services available. For instance, as discussed in Section 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 hierarchy of client profiles, and a description of the access rights that correspond to each profile, as shown in left-hand side of Fig. 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 us 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 adaptation intended by means of a *mapping* among the functionalities of two components being adapted. This adaptor specification will be used for the automatic construction of an *adaptor* that mediates their interaction.

Let us consider the specification of a possible *Client(c)* component for our VoD system, as represented by the session type *CLIENT* below:

```plaintext
CLIENT = ! !menu.
?info.
!( play. ?data. 0 + download. ?data. 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 (e.g. ?data vs. ?stream) and/or parameters in both components (omitted here for simplicity, according to the typing system in Fig. 2) 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 describing the intended connection between both components by means of sets of correspondences between their actions. Then, behavioural mismatch will be solved (if possible) by a process that builds the adaptor, as shown in Section 5.4.

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 non-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 adaptor specification expressing the adaptation required for our example consists of the following association rules:

```plaintext
S = { !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). Hence, the specification S establishes a correspondence between input/output actions in both components.
For instance, one !\text{menu} output action in the CLIENT is mapped to one ?\text{search} input action in the VoD (first rule in \(A\)).

The second rule in \(S\) maps ?\text{info} to !\text{list}. Notice here how the use of the name \text{string} in both terms of the rule makes explicit the correspondence between data parameters in the actions mapped. Parameter names have a global scope in the adaptor specification, 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 (!\text{play}) is mapped to two different actions in the VoD system (?\text{view} and ?\text{start}). The same occurs in the fourth rule, now with the action !\text{download}. Moreover, these two rules establish a non-deterministic correspondence between VoD’s input action ?\text{view} (which appears in both of them), and either client’s !\text{play} or !\text{download} output actions.

The fifth rule contains again a one-to-many correspondence between actions. It states that whenever the system issues a !\text{stream} action, a ?\text{data} action will be performed by the client (transmitting this way the video data), but also that a confirming ?\text{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 \(S\) indicates that VoD’s action \text{quit} has no correspondence in the client, so that it may be matched by the adaptor whenever the VoD system requires it.

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

5.2. Definition of adaptor

An adaptor specification defines the behavioural properties that an adaptor component must satisfy. Each rule in an adaptor specification can be (automatically) translated into a term in the process calculus \(L\) that defines a certain property. For instance, the semantics of the first rule in \(S\) is given by the term \(m1\) below:

\[ m1(1,r) = 1?\text{menu}(). \ ( r!\text{search}(""). 0 || m1(1,r) ) \]

which indicates that whenever the component on the left (here the CLIENT, represented by the session 1) performs a !\text{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 at that particular moment — perform a ?\text{search} action with an empty string in place of the missing title.

Similarly, the rest of the rules in \(m\) can be translated into processes in \(L\). Then, for a given adaptor specification \(S\), we will denote by \(\Pi(S)\) — the semantics of the specification — the parallel composition of the properties defined by the rules in \(S\).

We can now define the processes that satisfy an adaptor specification \(S\) as the set of processes which are simulated by the process \(\Pi(S)\). Notice that, in general, this is an infinite set of processes. We consider a standard notion of (strong) simulation.

Hence, the adaptor specification \(S\) provides a minimal description of an adaptor that plays the role of “component-in-the-middle” between the components being adapted, and mediating their interaction. It is worth noticing that an adaptor specification 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 Section 5.4. 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:

\textbf{Definition 1 (Adaptor).} Let \(\alpha_P\) and \(\alpha_Q\) be session types for two components \(P\) and \(Q\), respectively, and let \(S\) be an adaptor specification. A process \(A\langle 1,r \rangle\) is an \textit{adaptor} for \(\alpha_P\) and \(\alpha_Q\) under \(S\) iff:

(1) \(A\langle 1,r \rangle\) satisfies \(S\), and

(2) \([A]\satisfies \alpha_P\) and \([A]_r\satisfies \alpha_Q\).

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.

As we will show in Section 5.4, given an adaptor specification, and the session types of the components being adapted, an automatic procedure 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.

Hence, the adaptor generation process not only takes into account the adaptor specification \(S\), but also the session types representing the behaviour of the components being adapted, in order to explore this set \(\Pi(S)\) for finding an actual adaptor for these components.

5.3. Safe composition

As we already mentioned, the conditions to be fulfilled by an adaptor guarantee the safe interaction of the components being 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 \(P \xrightarrow{*} E \rightarrow\), we have that either:

1. \(E \equiv 0\), or
2. if \(E \xrightarrow{\xi} \) then \(\xi = l\text{acp}(s)\) or \(\xi = l\text{rqt}(s)\) for some link \(l\) and some session \(s\).

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 deadlock. We now present a result that states that the definition of an adaptor ensures the conditions for guaranteeing that the interactions are safe, thus establishing a correspondence between a property on session types (compatibility), and the way the corresponding processes proceed. The Theorem 3 below states that a deadlock-freedom result for the parallel composition of two processes can be deduced from the compatibility of the corresponding session types. The proof of this and other intermediate results can be found in [6]. Of course, this result is only derivable when processes do not interact (among themselves or with others) through other sessions different from those considered for adaptation.

Theorem 3. Let \(P, Q\) be three processes sharing only two links \([l, r]\) such that \(\text{fn}(P) \cap \text{fn}(Q) = \emptyset\), and for every session \(s \notin \{s_l, s_r\}\) we have \([P]_s = [A]_s = [Q]_s = \perp\). If \(A\) is an adaptor for \([P]_l\) and \([Q]_r\) under a certain specification \(S\), then \(P || A || Q\) is session safe for \([l, r]\).

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 \(s_l\) and \(s_r\).

5.4. 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 an adaptor specification \(S\). Given such a specification \(S\), and the session types \(\text{VoD}\) and \(\text{Client}\), 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 \(\text{VoD}\) and \(\text{Client}\) as stated in the adaptor specification, 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 \(\text{Client}\) and \(\text{VoD}\), and not from the component implementation represented by the processes \(\text{Client}\) and \(\text{VoD}\).

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 an adaptor specification specifying the intended adaptation between them, the adaptor returned will be a component-in-the-middle such that:
The parallel composition of the adaptor with the components being adapted will not deadlock, and
the adaptor will satisfy the action correspondences and data dependencies stated by the adaptor specification, by
possibly introducing some subservicing according to the access rights of the components involved.

Due to its inherent nondeterministic nature, the algorithm has been implemented in Prolog. We focus here 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
subservices 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/3) that such process satisfies the (expanded) adaptor specification.

\[
\text{adapt}(\text{interface}(Tp, Dp), \text{interface}(Tq, Dq), \text{Spec}, \text{Adaptor}) :-
\text{transform_spec}(\text{interface}(Tp, Dp), \text{interface}(Tq, Dq), \text{Spec}, \text{XSpec}),
\text{find_adaptor}(\text{XSpec}, \text{par}(Tp, Tq), 0, \text{nil}, \text{Adaptor}),
\text{satisfy}(\text{Adaptor}, \text{XSpec}, \text{NewSpec}).
\]

Predicate find_adaptor/5 basically implements a loop which tries to incrementally build an adaptor by
progressively eliminating all the deadlocks that may occur in the interaction of the adaptor with two components
represented by the session types Tp and Tq. Notice that since the conditions of Theorem 3 establish that each of
the components being adapted shares only one link with the adaptor, we can reconstruct a hypothetical process
specification of the components being adapted from their types, and check these processes for deadlocks with
the adaptor being generated. 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, otherwise it
fails.

\[
\text{find_adaptor}(_, \text{Context}, \text{Adaptor}, _, \text{Adaptor}) :-
\text{deadlocks}(\text{and}(\text{Context}, \text{Adaptor}), []).
\]

\[
\text{find_adaptor}(\text{Spec}, \text{Context}, \text{Adaptor}, \text{Last}, \text{NewAdaptor}) :-
\text{deadlocks}(\text{and}(\text{Context}, \text{Adaptor}), [\_\_]),
\text{states_after_last}(\text{and}(\text{Context}, \text{Adaptor}), \text{Last}, \text{States}),
\text{deadlocks}(\text{States}, [\text{D}|\text{Ds}]),
\text{successes}(\text{States}, []),
\text{choose_unlocking_action}(\text{Spec}, [\text{D}|\text{Ds}], \text{Action}),
\text{add}(\text{Action}, \text{Last}, \text{Adaptor}, \text{Adaptor1}),
\text{find_adaptor}(\text{Spec}, \text{Context}, \text{Adaptor1}, \text{Action}, \text{NewAdaptor}).
\]

\[
\text{find_adaptor}(\text{Spec}, \text{Context}, \text{Adaptor}, \text{Last}, \text{NewAdaptor}) :-
\text{deadlocks}(\text{and}(\text{Context}, \text{Adaptor}), [\_\_]),
\text{states_after_last}(\text{and}(\text{Context}, \text{Adaptor}), \text{Last}, \text{States}),
\text{deadlocks}(\text{States}, []),
\text{prefix_of_last}(\text{Last}, \text{Adaptor}, \text{NewLast}),
\text{find_adaptor}(\text{Spec}, \text{Context}, \text{Adaptor}, \text{NewLast}, \text{NewAdaptor}).
\]

The first rule specifies that if there is no possible deadlock in the interaction of the adaptor with the components
represented by the two session types, 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 deadlock states.
Instead, when deadlocks may occur as a result of the interaction of the adaptor and the components, 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 \textit{states\_after\_last}, which collects in its third argument the set of states reachable by executing action \textit{Last}, and by predicate \textit{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 \textit{successes/2} is used to verify that no success states can be reached after executing \textit{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 \textit{Last}, then the algorithm has to remove \textit{Last} from the adaptor and backtrack, as any attempt to add a new action after \textit{Last} will unavoidably spoil those success states. If instead only deadlocked states (and no success state) can be reached after executing \textit{Last}, then an action \textit{Action} capable of unlocking one of these deadlocks is chosen non-deterministically and used to expand (predicate \textit{add/4}) the adaptor as one of the possible actions following \textit{Last}. The construction process continues then with a tail recursion, where \textit{Action} becomes the new last action added to the adaptor. Predicate \textit{choose\_unlocking\_action/3} is in charge of non-deterministically choosing an action \textit{Action} 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 access rights needed 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” \textit{Last}. The algorithm hence backtracks one step behind, by considering the action \textit{NewLast} that prefixes \textit{Last} in the Adaptor to continue recursively.

In order to illustrate how the algorithm works, let us consider our example again. 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 \textit{movies} is the access right corresponding to the \textit{CLIENT} component described in Section 4.2. The algorithm is executed with the predicate call:

\[ \text{?- adapt}(\text{IVod}, \text{IClient}, S, A). \]

where \text{IVod} and \text{IClient} represent, respectively, the session types \text{VOD} and \text{CLIENT} (together with their access rights and subservice definitions), \text{S} represents the adaptor specification, and \text{A} is the variable which will be instantiated to the adaptor constructed.

First of all, an extended specification (let us name it \text{XS}) of the adaptation required is constructed with the predicate \textit{transform\_spec/4}, taking into account the original specification \text{S} provided, and the subservice definitions in the interfaces. When a rule in \text{S} refers to an action for which a subservice is defined, the specification is extended with a new version of the rule, referring to the subservice instead. Hence, the resulting extended specification is:

\[ \text{XS} = \{ \text{!menu()} \leftrightarrow \text{?search(""}); } \]
\[ \text{?info(string)} \leftrightarrow \text{!list(string)}; \]
\[ \text{!play(title)} \leftrightarrow \text{?view(title), ?start();} \]
\[ \text{!play(title)} \leftrightarrow \text{?preview(title);} \]
\[ \text{!download(title)} \leftrightarrow \text{?view(title), ?record();} \]
\[ \text{!download(title)} \leftrightarrow \text{?view(title), ?start();} \]
\[ \text{!download(title)} \leftrightarrow \text{?preview(title);} \]
\[ \text{?data(video)} \leftrightarrow \text{!stream(video), ok();} \]
\[ \leftrightarrow \text{?quit();} \]

Then, the adaptor construction itself starts with a call to \textit{find\_adapter/5}. The adaptor will communicate with the components \text{Client} and \text{VoD} by means of two links, that we may call \text{client} and \text{vod}, respectively. At this initial point, the adaptor \text{A} is the empty process \text{0}, and the \text{Last} action added is \text{nil}. The predicate \textit{deadlocks/2} reports to two possible deadlocks in the interaction of the adaptor, the client, and the VoD service (represented by their corresponding session types): one is that the client deadlocks in a request action on link \text{client} (as indicated by the initial \text{!} in the session type \text{CLIENT}); the other one is that the VoD service deadlocks in an accept action on link \text{vod} (as indicated by the initial \text{?} in \text{VOD}). Since there are deadlocks after \text{Last}, the second rule of \textit{find\_adapter} applies. Let us suppose that first of these deadlocks is selected in \textit{choose\_unlocking\_action/3}, and the construction starts by expanding \text{A}:
A = client.accept?(c).

and Last points to this initial action in the new call to find_adaptor/5 that is issued. Since there are still deadlocks after Last, construction goes on the same way expanding the adaptor with a new action that now unlocks the deadlock transition accept in the VoD service:

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

Hence, we have established two sessions, that we have called c and v, the first one with the client and the second with the VoD. The creation of these sessions match the actions ! and ? in the session types of the components.

Now, the adaptor can be expanded with two different actions, each unlocking 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:


At this point, the session VoD ends, but the process of adaptation goes on with the session DAEMON caught 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 adaptor specification — 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 S, the adaptor gets the data parameter string to match the 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 until 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 extended adaptor specification. Since the VoD action !quit is mapped to no action, the adaptor may match this action freely when required by the VoD.

At this point, both the types CLIENT and DAEMON end. Thus, there are no more deadlocks after the last action added, and this branch of the adaptor construction is finished. However, since there are still deadlocks in the interaction tree of the components, the third rule of find_adaptor/5 now applies, and the construction backtracks to the point of the alternative between !play and !download. Then, this last action is matched, resulting in the adaptor:

A = ... c!info(string). ( c?play(title). ... d!quit(). 0


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 it 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 in a similar way, rendering at the end the full adaptor component:
A = client.accept?(c). vod.request!(v).
  ( 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 fulfills the extended adaptor specification XS (predicate satisfy/3). In this predicate, a new version S' of the specification is computed, taking into account which were the specification rules effectively used during adaptor construction (here, apart from those originally in S, only the sixth rule in XS, subservicing the ?record command for which the client did not have rights, with its subservice ?start):

\[
S' = \{ !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
\}

where the modified rule is marked with asterisks.

The returned adaptor A will allow the client and VoD components to interact without deadlocks, satisfying the access rights and subservice definitions, and fulfilling the adaptor specification S suggested by the client (except for the subservicing mentioned).

As a second example of adaptor generation, 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 at the point in which client actions !play and !download are to be matched. According to the adaptor specification 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:

   d!quit . c!data(video). 0

and the algorithm will return the adaptor A' together with the modified adaptor specification S'':

\[
S'' = \{ !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
\}

Finally, let us now consider whether the adaptors generated indeed satisfy Definition 1 and thus, whether we can expect a safe composition between the Client, the adaptor and the VoD system. The session types corresponding to the adaptor component A above are:

A_c = ? ?menu. !info.
     ?( play. !data. 0
       + download. !data. 0 )

A_v = ! ?(A_d). 0

A_d = !search. ?list
     !( view. !start. ?stream. !ok. !quit. 0
       + view. !start. ?stream. !ok. !quit. 0 )
and it can be easily proved both that (1) the adaptor satisfies the extended specification \( \Xi \) (i.e., it is simulated by \( \Pi(\Xi) \)), and also that (2) its session types are compatible with those of the client and the server:

\[
\text{CLIENT} \leadsto A_c \text{ and } A_v \leadsto \text{VOD}
\]

Hence, we are under the conditions of Theorem 3, and we can conclude that the system:

\[
\text{Client}(c) \parallel A(c,a) \parallel \text{Vod}(a,b) \parallel \prod_{i=1}^{n} \text{VoDDaemon}(b)
\]

is session-safe for the links \( \{a, c\} \), and now these components are able to interact successfully. Indeed, after the client performs its session with the VoD system, both the client and the adaptor end, and the VoD system and its daemons are again in their original states,

\[
\text{VoD}(a,b) \parallel \prod_{i=1}^{n} \text{VoDDaemon}(b)
\]

all of them expecting to perform an accept action, requested by a new client (that may need a completely different adaptation).

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, defined in [23,15,16,6], 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 fully-fledged process algebras, while featuring an expressiveness bonus w.r.t. the approaches based on finite state machines [16].

It is also interesting to relate our notion of subservicing to the concept of action refinement in process algebras [2,12]. 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 always be performed, subservices express more general and adaptive 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 [22], where an extension of the \( \pi \)-calculus is proposed to express compositions. However, the focus of [22] is different from ours. While our goal is the automatic deployment of adaptors, [22] 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. [1,11]), 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.

Acknowledgements

We would like to thank the reviewers for their helpful suggestions and comments on early drafts of this paper.

References