Soft component adaptation

Antonio Brogi [1,4]

Dipartimento di Informatica, Università di Pisa, Italy

Carlos Canal [2,5] Ernesto Pimentel [3,5]

Dpto. de Lenguajes y Ciencias de la Computación, University of Málaga, Spain

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 behaviours. The notions of access rights (associating components with the services they are allowed to use), and sub-servicing (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 [6,8]. The possibility for application builders to easily 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 [5]. 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

[1] Email: `brogi@di.unipi.it`
[2] Email: `canal@lcc.uma.es`
[3] Email: `ernesto@lcc.uma.es`
[4] The work of A. Brogi has been partly supported by MIUR Project NAPOLI.
[5] The work of C. Canal and E. Pimentel has been partly supported by the projects TIC2002-4309-C02-02 and TIC2001-2705-C03-02, respectively, funded by the Spanish Ministry of Science and Technology

between components in the perspective of adapting or wrapping them to overcome such differences. However, even if all signature problems may be overcome, there is no guarantee that the components will suitably interoperate as mismatches may also occur at the protocol level, because of the differences in the protocols defining the behaviour of the components involved [14]. 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 [3], we have developed a formal methodology for component adaptation that supports the successful interoperation of heterogeneous components presenting mismatching interaction behaviours. The main ingredients of the methodology can be summarised as follows:

(i) *Component interfaces.* IDL interfaces are extended with a formal description of the behaviour of the components, which explicitly declares the interaction protocol followed by a component.

(ii) *Adaptor specification.* Adaptor specifications can be simply expressed by a set of correspondences among actions and parameters of the two components. The distinguishing aspect of the notation is that it produces a high-level, partial specification of the adaptor.

(iii) *Adaptor derivation.* A concrete adaptor is fully automatically generated, given its partial specification and the interfaces of two components, by exhaustively trying to build a component which satisfies the given specification. The separation of adaptor specification and derivation permits the automation of the error-prone, time-consuming task of constructing a detailed implementation of a correct adaptor, thus simplifying the task of the (human) software developer.

A limitation of the adaptation technique described in [3] 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, a soft adaptor could be nevertheless deployed by weakening some of the requirements stated in the specification.

In this paper, we shall extend the methodology for component adaptation presented in [3] precisely to overcome this type of limitation. The notions of access rights (associating components with the services they are allowed to use), and sub-servicing (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 shall rely on *session types* (rather than on π-calculus as in [3]), firstly defined in [9] to describe (possibly non-terminating) component behaviours as true types.

It is worth outlining that while we shall use the notion of access rights to enforce a secure adaptation of component services, we will not deal here with other important aspects of security, such as authentication protocols,

which will have to be additionally employed to enforce a secure communication between the components.

The rest of the paper is organised as follows. In Sect. 2 we introduce a simplified example of Video-on-Demand service, which will be used throughout the paper to illustrate the methodology. Session types are introduced in Sect. 3, and their use to describe the VoD 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 behaviours. Finally some concluding remalrs 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 remote clients to a database of movies and news in video format.

There are four different profiles of clients. Each profile grants certain access rights to the session. 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.

First, *guests* may *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 news. The *movies* profile grants access to *view* movies but not news, while *full* clients may access to both news and movies. Moreover, while viewing a movie, *full* users may also *record* it permanently in their computers.

Notice that, 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 *play* 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 above mentioned profiles, depending on the identity of the client, and generates (if necessary) an adaptor for connecting the client 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 by 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 the interfaces both of the client and its own, constructs an adaptor component.
- If during the construction the adaptor specification given by the client cannot be fully satisfied, the system returns a modified adaptor specification that must be accepted by the client before using it for their interoperation.

3 Session types

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 [2,4,10,11].

The advantage of having these formal specifications of components is twofolded. On the one hand, the component behaviour is unambiguously fixed and it can be animated with a convenient interpreter tool. A second important advantage is the possibility of analyzing a number of (liveness and security) properties such as safe composition or replacement in complex systems, or how to adapt incompatible behaviours. In spite of the usefulness of process algebras for component descriptions, they present an important drawback due to the complexity of the decision procedures to verify the mentioned properties. In order to cut this complexity, we have applied to our context the notion of *session types* introduced in [9].

A session type is a chain of dyadic interactions whose collection constitutes a program. Each session is designated by a private link, through which interactions belonging to that session are performed. The use of dyadic 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 behaviors 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 process calculus $\mathcal{L}$ and a session type description language. The former is simply used as a way of referring to the actual implementation of the components, whereas the latter is used to type the behaviour of the components.

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

$$P ::= 0 \;\mid\; A.P \;\mid\; \sum_i k!A_i.P_i \;\mid\; \sum_i k?A_i.P_i$$

where 0 represents the empty process, k denotes a session name, and A,A_i denote messages syntactically composed by a message selector and a sequence of arguments. We consider two kinds of actions: Output actions, where a message is sent through a session or link, and input actions where messages are received from a link, respectively:

$$k!message(args) \qquad k?message(args)$$

Process expressions	The type of session k
$a!request(k).\ P$	$!\ P^{(k}$
$a?accept(k).\ P$	$?\ P^{(k}$
$k!throw(k').\ P$	$!(k')\ P^{(k}$
$k!catch(k').\ P$	$?(k')\ P^{(k}$
$\sum_i\ k!A_i.\ P_i$	$!\sum_i\ A_i.\ P_i^{(k}$
$\sum_i\ k?A_i.\ P_i$	$?\sum_i\ A_i.\ P_i^{(k}$
$P \mid Q$	$P \odot Q$
0	0

Table 1
Session types for processes.

We will distinguish four special message selectors: *request*, *accept*, *throw*, and *catch*, all of them with a single argument representing a session name. Messages corresponding to *request* and *throw* are always output actions, and they will only appear in expressions like:

$$a!request(k) \qquad a!throw(k)$$

Similarly, *accept* and *catch* are considered input actions:

$$a?accept(k) \qquad a?catch(k)$$

A *request* action issued on a channel a waits for the acceptance (*accept*) of a session on this channel. When these two actions synchronize, a new session is created linking the processes where the two actions were invoked. 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 must be made to capture the session). These last two complementary actions permit to dynamically change the architecture topology.

Our interest is not however focussed on using a process calculus like $\mathcal{L}$ for describing the behaviours of software components, but rather in *typing* these behaviours. Therefore we will employ behavioural type specifications as the basis for establishing the correct interaction among the corresponding components. In order to type behaviours, we provide a type system derived from the notion of session types presented in [9]. We define a type for each expression in $\mathcal{L}$, as indicated in Table 1, where $P^{(k}$ denotes the type of session k in process P.

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, using the corresponding data types instead;
- session types are limited to binary communications between two components;
- mobility is expressed by means of *throw*/*catch* actions, and since sessions are binary, once a process throws a session, it cannot use it anymore;
- parallel composition is restricted within session types, the parallel composition of processes is represented by the binary composition ($\odot$) of their corresponding session types;
- no mixed alternatives are allowed: Input and output actions cannot be combined in a single alternative, and alternatives (either input or output) cannot be made on different channels.

Notice however that these are not relevant limitations in our context, as we will see below. On the contrary, these restrictions make session types a calculus much more tractable than other studied alternatives, like π-calculus or CSP. The advantages of employing session types instead of other concurrency formalisms are thoroughly discussed in [15].

4 Service specification

We now provide a specification of the VoD service described in Sect. 2 both in terms the process calculus $\mathcal{L}$ and of session types. Actually, $\mathcal{L}$ is used here just as a form of implementation, to show the actual components the session types refer to. Indeed component specifications will be normally provided by session types, which can be derived from those process implementations.

4.1 Behaviour of the VoD service: Process algebra

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

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

Once the session daemon `VoDDaemon` accepts the session opened by the `VoD` system [6], it is ready to accept different commands from the client. Each command implies a certain sequence of messages to be followed (i.e., a protocol).

[6] Although not necessary, we use the same parameter names (e.g., `b`) in different processes to help intuition.

For instance, after selecting a movie with the `view` command, the client mayt either issue a `start` or a `record` command to start visualization or recording, while other commands like `preview` or `news` apply immediately. 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 information is performed by `VoDStream` via an output action `stream` which must be replied with `ok` for indicating a correct reception of the data, or with `retry` for repeating the 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: Session types

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 session types corresponding to these processes with respect to their interaction with the clients through links `vod` and `daemon` —that is the session types $\texttt{VoD}^{(\texttt{vod}}$ and $\texttt{VoDDaemon}^{(\texttt{daemon}}$. For short, we will call them `VOD` and `DAEMON`, respectively.

These session types will be used as the specification of the VoD service, in order to derive adaptors for connecting clients to the system. Accordingly to Table 1, the type of each session can be automatically derived from the corresponding process descriptions. 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 components behaviour, and the interactions between the `VoD` process and the `VoDDaemon` are not shown in the `VOD` session type, 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 both in the process calculus and in the corresponding session types, the behaviour of the service was described completely independently of access rights and subservices, thus following the principle of separation of concerns typical of aspect-based programming. The motivation here is to present the definition of access rights and subservices as a way of configuring the system: The same 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 interactive behaviour of the system when a session is opened by a client. However, this information is not enough to safely connect client components to the VoD.

In particular, session types do not include information on the access rights that corresponds to the available services. For instance, as discussed in Sect. 2, a *view* action should only be available to clients with a *movies* or *full* profile. This information must be provided by the component as part of its IDL 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 Figure 1. Obviously, actions that are accessible to a given profile are also accessible to those higher in the graph.

If a client opens a VoD session and requests a service without owning the appropriate access rights, the system will typically reply by raising an exception without providing the service. However, in many cases such a behaviour may be considered too strict, and a more flexible behaviour would be desirable. For instance, the system could provide a different service (accessible by

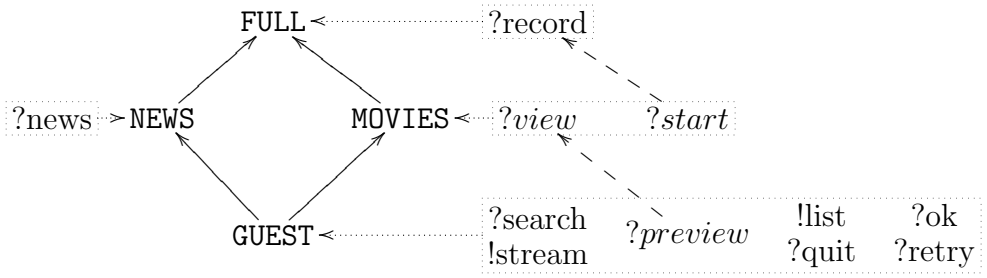

Fig. 1. Client profiles (solid arrows), access rights (dotted arrows), and subservices (dashed arrows) in the VoD system.

the client) as an alternative to the one requested.

It is worth observing that our approach allows to feature such flexible adaptations without having to modify and complicate protocol specifications. Indeed component interfaces are extended so as to include information about subservices, so that some of the services offered by the component can be substituted by a subservice. 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. 1. 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 to be adapted. This description will be then used for the automatic construction of an *adaptor* that mediates the interaction of the two components.

Let us start by the specification of a possible client of our VoD system, as represented (directly) by the session type `CLIENT` below [7]:

```
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 it 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`, and then end.

[7] We omit the description in the process calculus $\mathcal{L}$ of the actual implementation of the `Client` component since it would be very similar to the corresponding type `CLIENT` shown here.

The mismatch between the specification of the VOD system and that of its CLIENT is both *syntactic* and *behavioural*. Syntactic mismatch deals with discrepancies in action names in both components (e.g., ?data and ?stream), and also with parameter rearrangement, while behavioural mismatch deals with protocols and command ordering (for instance, the fact that the client assumes speak directly with 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 the VoD system and its client. Then, behavioural mismatching will be automatically solved (if possible) by an automatic 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 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 mismatch where, for instance, reordering and remembering of messages is required. The adaptor will be specified by means of a *mapping* that establishes a number of rules relating actions and data of two components. For instance, the mapping expressing the intended adaptation for our example consists of the following 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
      NONE              <> ?quit();                  // 6th
    }
```

where, as a convention, all the actions on the left hand side 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).

The intended meaning of the first rule of M is that whenever the client performs a !menu command, the VOD will eventually (not necessarily at that particular moment) perform a ?search input action with an empty string in place of the missing title in the left part.

Similarly, the second rule maps ?info to !list. Notice here how the real parameter name 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 the same 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 (e.g. !play) is mapped to two different actions in the VoD system: ?view *and* ?start. The same occurs in the forth 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 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 (NONE) in the client, so that it may be matched by the adaptor whenever the VoD system requires it (though it will not be transmitted to the client, since no correspondence is given to this particular action).

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 a minimal specification of an adaptor that will play the role of a "component-in-the-middle" between the VoD and the client, mediating their interaction. It is important to observe that the adaptor specification defined by a mapping abstracts from many details of the component behaviours. The burden of dealing with these details is left to the (automatic) process of adaptor construction, that will be described in Sect. 5.2 below.

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 that solves the syntactic mismatching between both components. 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, while it must also solve all behavioural mismatches between the actual protocols followed by the two components.

Notice that the adaptor is a process (i.e. an actual 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 `Client` and `VoD`.

Space limitations do not allow us to present here the algorithm for soft adaptor derivation in full detail, but a detailed description of the basic algorithm can be found in [3]. The major differences are that now the algorithm takes also into account access rights and subservice definitions, and that when the mapping `M` cannot be fully satisfied, an adaptor will be generated anyway, by satisfying the mapping as much as possible. We therefore speak of *soft* adaptors, instead of strict or *hard* adaptors as in [3]. We shall however summarise the essence of the algorithm. Roughly speaking, the adaptor will be a *component-in-the-middle* `a` such that:

- The parallel composition `( Client | A | VoD )` will not deadlock,
- `A` will ensure that access rights are not violated, possibly offering subservices when needed, and
- `A` will satisfy *as much as possible* the action correspondences and data dependencies specified by the mapping `M`.

To achieve the three conditions above, the algorithm tries to build incrementally an adaptor process `A` by progressively eliminating all the deadlocks that may occur in the evolution of `(Client|a|VoD)`. The algorithm is basically a loop which keeps track of the adaptor `A` constructed so far. While the parallel composition `(Client|a|VoD)` is not deadlock-free, the algorithm tries to extend `A` with an action that will trigger one of the deadlock states. The trigger action is chosen in a way that it satisfies both the dependencies between actions and their data parameters indicated in the mapping, as well as the access rights and subservicing definitions. Because of its inherent nondeterministic nature, the construction has been naturally implemented in Prolog.

To grasp the idea of 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 for 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` are Prolog terms representing the session types `VOD` and `CLIENT`, `m` is a term representing the mapping `M`, and `A` is the variable which will be instantiated to the representation of the constructed adaptor.

First of all, the adaptor will communicate with the components `Client` and `VoD` by means of two links, that we call `client` and `vod`, respectively.

The construction starts by establishing two sessions —let us call them `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:

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

The adaptor can be now expanded with two different actions, each triggering a deadlock in the components. These actions are `v?(d)`, catching a session `s` to match `VOD`'s action `!(DAEMON)`, on the one hand, or `c?menu()`, matching `CLIENT`'s action `!menu`. Eventually, the adaptor will be expanded with these two actions, the order not being relevant. Suppose that the adaptor is expanded into:

```
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 in 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 action selected, and the adaptor expands into:

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

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

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

The `CLIENT` session is now deadlocked in the 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(titile)`, 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 session type `VOD`.

At this point, both the types `CLIENT` and `DAEMON` end, so 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!view(title). ... . 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 substitute `?start` as offered instead).

Hence, the algorithm will return the adaptor `A` together with a notification of modification of the mapping due to access rights and subservices definition in the VoD system:

```
      !download(title)   <> ?view(title), ?start();    // 4th
```

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

As a final example, let us suppose now that the profile of the client component was `guest`, which would not allow her to view movies. The call for constructing the adaptor would be in this case:

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

and the construction of the adaptor would be more or less the same 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 in this case the constructed adaptor `A'` together

with the modified mapping rules:

```
!play(title)       <> ?preview(title);      // 3rd
!download(title)   <> ?preview(title);      // 4th
```

6 Concluding remarks

In this paper, we have tried to illustrate the main aspects of a formal methodology for the automatic development of adaptors capable of solving behavioural mismatches between heterogeneous interacting components. The proposed methodology extends the adaptation technique described in [3] 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, and access rights and subservice declarations, respects the separation of concerns advocated by aspect-based programming, 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. More specifically, we adopt the approach of enriching component interfaces with behavioural descriptions as in [2,7,12], to cite but a few of the more closely related works. A distinguishing feature of our approach consists of adopting *session types*, firstly defined in [9], to describe (possibly non-terminating) component behaviours as true types. The adoption of session types sensibly reduces the complexity of verifying properties w.r.t. other approaches based on full-fledged process algebras (e.g., [12]), while featuring an expressiveness bonus w.r.t. the approaches based on finite state machines (e.g., [16]). A thorough comparison of our adaptation methodology with other proposals is discussed in [3], while a comparison between session types and other formalism is reported in [9].

The issue of assembling off-the-shelf components using *wrappers* to encapsulate components and enforce security policies has been recently addressed in [13], where an extension of the π-calculus is proposed to express compositions and a causal type system is used to prove properties of given wrappers. While the focus of [13] differs from ours in that we focus on the semi-automatic deployment of adaptors, the ultimate objectives are similar and we plan to investigate further the relations between the two approaches.

Finally, as we already anticipated in the Introducion, 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] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, 1997.

[3] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software, Special Issue on Automated Component-Based Software Engineering*, 2003. (in press). A preliminary version of this paper was published in *Component deployment*, LNCS 2370, pages 185–199. Springer, 2002.

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

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

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

[7] D. Compare, P. Inverardi, and A.L. Wolf. Uncovering architectural mismatch in component behavior. *Science of Computer Programming*, 33(2):101–131, 1999.

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

[9] Kohei Honda, Vasco T. Vasconcelos, and Makoto 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.

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

[11] 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. Springer, 1995.

[12] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Software Architecture*, pages 35–49. Kluwer, 1999.

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

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

[15] 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.

[16] D.M. Yellin and R.E. Strom. Protocol specifications and components adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.