Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 175 (2007) 81–95
www.elsevier.com/locate/entcs

Dynamic Contextual Adaptation[1]

Antonio Brogi[a], Javier Cámara[b], Carlos Canal[b], Javier Cubo[b] and Ernesto Pimentel[b]

[a] *Dept. of Computer Science, University of Pisa, Italy.*
Email: brogi@di.unipi.it

[b] *Dept. of Computer Science, University of Málaga, Spain.*
Emails: jcamara@lcc.uma.es, canal@lcc.uma.es, cubo@lcc.uma.es, ernesto@lcc.uma.es

Abstract

When developing systems based on COTS, components need to be adapted in most of the occasions to work under certain conditions which were not initially predicted by their developers. Thus, it is very important to provide systems with the skill to dynamically alter their behaviour while running, depending on the changing conditions of the environment. In this work we describe a context-dependent, dynamic mapping between the interfaces of the components being adapted, overcoming some of the limitations of the static mappings presented in previous works. This is achieved by means of *contextual environments*, which define flexible adaptation policies. We also present a case study, illustrating the proposal, and discuss the improvements these *mappings* represent in comparison with previous works, as well as some open issues.

Keywords: Adaptation, dynamic, flexible, component, contextual environment, mapping, behaviour, interoperability.

1 Introduction

One of the most significant trends in the software development area is that of building systems incorporating pre-existing software components, commonly denominated *commercial-off-the-shelf* (COTS). These are stand-alone products which offer specific functionality needed by larger systems into which they are incorporated. The notion of developing a system by writing code is being replaced by the assembly of existing components, which in conjunction achieve the desired functionality. However, it turns out that the constituent components often do not completely fit one another when reused, and adaptation has to be performed to eliminate the resulting mismatches [2]. Therefore, software composition always requires a certain degree of adaptation [12] in order to ensure that conflicts among components are

[1] This work has been partly supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Science and Technology (MCYT).

doi:10.1016/j.entcs.2007.03.005

minimised and that the desired functionality is achieved. The need to automate the aforementioned adaptation tasks has driven the development of Software Adaptation [7], a new approach characterized by highly dynamic run-time procedures that occur as devices and applications move from network to network, modifying or extending their behaviour, thus enhancing the flexibility and maintainability of systems. Software Adaptation promotes the use of software adaptors [14], which are software entities capable of enabling components with mismatching behaviour to interoperate at four different levels:

Signature Level: Interface descriptions at this level specify the methods or services that an entity either offers or requires. These interfaces provide names, type of arguments and return values, or exception types. This kind of adaptation implies solving syntactical differences in method names, argument ordering and data conversion.

Protocol Level: Interfaces at this level specify the protocol describing the interactive behaviour that a component follows, and also the behaviour that it expects from its environment. Indeed, mismatch may also occur at this protocol level, because of the ordering of exchanged messages and of blocking conditions. The kind of problems that we can address at this level is, for instance, compatibility of behaviour, that is, whether the components may deadlock or not when combined.

Service Level: This level groups other sources of mismatch related with non-functional properties like temporal requirements, security, etc.

Semantic Level: This level describes what the component actually does. Even if two components present perfectly matching signature interfaces, follow compatible protocols, and present no mismatch at the service level, we have to ensure that the components are going to behave as expected.

Adaptors already available in different component-oriented platforms like CORBA, J2EE or .NET address several adaptation issues at the signature level, allowing a certain degree of interoperability between software components. Indeed, they provide convenient ways to describe signatures using Interface Description Languages (IDLs), but they offer a quite limited and low-level support to describe the concurrent behaviour of components, since solving all signature problems does not guarantee that the components will interoperate properly. Recent research efforts [1,6,11] concentrate on the interoperability of reusable components at the behavioural level. This work deals with the problem of adapting mismatching behaviour that components may exhibit in a dynamic way, particularly tackling the signature and protocol levels. Regarding mobile and pervasive computing, this proposal will also enable infrastructures supporting automated software adaptation by extending applications at run time to adapt behaviour in a context-sensitive way as devices cross different networks and the environment changes.

In previous works [4], we focused on component adaptation both at the signature and protocol levels by establishing a correspondence or mapping between the messages exchanged by the components and deriving adaptors using that information in order to adapt mismatching behaviour. These mappings were defined by means of static message bindings (the messages were always mapped in the same

way between components, independently of the running conditions of the system). However, it would be desirable to map messages establishing a different correspondence depending on the situation. For instance, if we have a service that offers different features depending on some access rights to its users, adaptation should consider these rights by establishing appropriate correspondences between service requests and the subsequent responses provided by the service, discriminating users according to the permissions they have been granted. This idea led to a second approach described in [5] where this is achieved through the notion of *subservicing*. The idea was featuring a secure, *soft* adaptation of third-party software components when the given adaptation requirements could not be fully satisfied. Technically this was achieved by exploiting the notion of *subservice* (substitution of a service for another one which features only a limited part of its functionality) to suitably weaken the initial specification when needed. However, this proposal did not allow to describe the conditions under which a given service or subservice had to be used. So the adaptor obtained through this approach was underspecified. What if access rights change dynamically?

Our current proposal focuses on defining flexible adaptation policies by means of contextual environments, providing an adaptor with the ability to dynamically alter its behaviour during its execution depending on the (changing) conditions of the environment, so that the mismatching at the protocol level is addressed in this approach. When dealing with this broader scenario in previous works, the dynamically changing conditions of the system could not be controlled properly since the provided adaptation was static or immutable. This work hence represents a noticeable enhancement to our approach to adaptation.

The structure of this paper is the following: In Section 2, a motivating example is presented in order to clarify the usefulness of dynamic adaptation. In Section 3, subsection 3.1 describes our proposal, and a formal notation defining it is sketched. We continue the discussion of our example in 3.2 by describing a contextual mapping for it and comparing it to a static version presented in previous proposals, highlighting the enhancements this current proposal represents. In 3.3 a brief description of the semantics associated to this approach is given along with a discussion on how to derive a suitable adaptor for contextual adaptation. Finally, Section 4 draws up the main conclusions of this paper and sketches some future tasks that will be accomplished to extend its results.

2 Motivating Example

We present a simple case study in order to illustrate our proposal, and discuss the improvements this work purports in comparison with previous ones. The example consists on a simplified *Video-on-Demand* (VoD) system taken from [5], which is a Web service providing access to a database of movies. In Table 1, we present the VoD behavioural specification and the client interface in terms of a process algebra (pseudo-CCS). Although there are other alternatives for the representation of behavioural interfaces, such as Labelled Transition Systems (LTS) [8] or Finite

State Machines (FSM) [13], we have chosen process algebras because they allow the specification of behavioural interfaces concisely, and at a higher level. The issue of typing component behaviour and service specification applied have been discussed in recent works [1,9,10]. For the example, we assume a typical scenario where a *Client* component wishes to use some of services offered by the *VoD* system.

VoD : *behaviour specification*
$VoD = login?Guest$
$Guest = preview?Guest$
$\quad + subscribe?Subscriber$
$\quad + logout?VoD$
$Subscriber = view?Subscriber$
$\quad + download?Subscriber$
$\quad + unsubscribe?Guest$
$\quad + logout?VoD$
$Client$: *behaviour specification*
$Client = hello!Client'$
$Client' = play!Client'$
$\quad + record!Client'$
$\quad + switch!Client'$
$\quad + quit!0$

Table 1
VoD service specification: server and client behavioural interfaces.

There are two different user profiles on the VoD system depending on certain access rights:

- *Unregistered user or Guest.* These users are only allowed to preview movies available within the VoD catalogue.

- *Registered user or Subscriber.* These users are those who have paid a subscription fee, and are allowed a variety of services such as viewing any movie from the catalogue or downloading a copy of it into their computers.

A client session running on the VoD server can switch between these two profiles as the client subscribes (paying the corresponding fee) or unsubscribes from the service. A client session starts always with *Guest* as initial profile.

When a client opens a session with the VoD system, it may perform any permitted action (*play*, *record*), or else may dynamically request the change of its access rights to the server (*switch*). Finally, the client ends its session (*quit*). Note that

the client does not need to be aware of profiles or of user privileges. Thus, the client interface is always the same for the different types of users.

On the other hand, the VoD server waits for an incoming connection. Once a client connects issuing a *hello* message (received in the server as *login*), a session is opened. When a client opens a session with the VoD system, it follows a connection procedure which associates the initial profile with *Guest*. As we have already mentioned, the user profile may be changed during interaction between the server and a client (we want to get an adaptation between both components). Depending on the current profile, the server offers a variety of services to the client:

- *Guests*: When the client performs *play* or *record* actions, the server only allows the user to view a clip of the movie (*preview*). If the client issues a *switch* message to the server, this may *subscribe* the user to the service.

- *Subscribers*: a *play* action is interpreted by the server as a *view* message. In the case of *record*, the server allows users to *download* the movie into their computers. If the client issues a *switch* message to the server, this may *unsubscribe* the user from the service.

Finally, the client ends its session (sending a *quit* message to the server, which corresponds to *logout*). Once this session has finished, the VoD server waits for a new client connection.

Following this example, we observe that, apart from solving both syntactical and behavioural mismatch, adaptation has to distinguish between the two available user profiles when translating the messages to the VoD server. Using a static approach, message correspondences are fixed, so independently of the current user profile, a *record* message, for instance would have to be always translated to the same target message (either *preview* or *download*). This motivates the need to provide new capabilities for adaptation in order to achieve context-aware message translation. In our proposal, we advocate for the use of contexts for adaptation. In the example, the two user profiles available will be associated to two different contexts in the system, being able to change between them through certain actions (*switch* in this case study).

3 Proposal Description

3.1 Contextual Mappings

This proposal intends to overcome the limitations of previous ones by solving some issues concerning dynamic changes that may occur while the system is running. This will be achieved by providing a *contextual mapping* for the exchanged messages between components, enabling the system to cope with the situations where conditions are dynamically altered at runtime, obtaining an adaptation where messages are not necessarily translated in the same way depending on the current situation of the system.

If previously messages where mapped to other message(s) on the target component, in a contextual mapping messages are mapped to other messages as well, although they may also trigger a change of environment in which the specification of

the mapping being used will be different. This enables the extension or modification of the adaptation as the system is running.

We define the set of *contextual environments* as:

$$(\gamma \in)\Gamma = L_\perp \longrightarrow 2^R \cup (2^R \times \Gamma) \cup \perp$$

where L represents the alphabet of the source component (i.e., its methods), and R represents the one for the target component. An environment is a partial function, so there are actions of L for which there is not a specific environment defined. Instead of using just L, the function takes as domain $L_\perp$ allowing the mapping of actions in the target component without requiring the source component to issue any specific message. The function returns elements of $2^R, 2^R \times \Gamma$, or $\perp$ as image (note that this corresponds to a recursive definition). For instance, an environment could be defined as follows:

$$\begin{aligned}\gamma = \{\ & a \to 1; \\ & b \to 2, \{\ a \to 3;\ \ b \to 4\ \}; \\ & c \to 5\ \}\end{aligned}$$

Here, γ is a contextual environment that maps a to 1, b to 2 (and triggers a transition to a new context where a is mapped to 3, and b to 4), and c is always mapped to 5. The correspondences that prompt a change of context are separated by comma, and the rest of them by semicolon. We consider that all the values in the domain of the contextual environment which are not mapped to any value correspond to *bottom* ($\perp$). Based on the notion of contextual environment, the set of *mappings* is defined as:

$$(\mu \in)M = \Gamma \longrightarrow \Gamma$$

Thus, *mappings* are functions taking a contextual environment and returning another contextual environment. Note that $u_\perp \in \Gamma$ (whose textual representation is $\{\}$) denotes the *undefined* function ($\forall\, a \in L_\perp : u_\perp(a) = \perp$), since a contextual environment may be empty (it defines no correspondences between messages). A simple mapping may be, for instance:

$$\begin{aligned}\mu = \lambda\gamma.\{\ & a \to 1; \\ & b \to 2, \{\ a \to 3\ \} \rhd \gamma; \\ & c \to 4\ \}\end{aligned}$$

Here, μ is a mapping which transforms any environment γ in a new one where a is translated to 1, b to 2 (also triggering a transition to a new context where a is translated to 3, *overriding* ($\rhd$) the initial environment γ), and c is always translated to 4. We define the *override* operator as:

$$(\gamma \rhd \gamma')x = \begin{cases} \gamma' x & \textit{if } \gamma x = \perp \\ \gamma x & \textit{otherwise} \end{cases}$$

Our proposal for contextual mapping is partially inspired by the module calculus defined in [3], where overriding and also other similar operations are defined. The notation we use here for mappings enables adaptation to change message translation

dynamically, depending on the current context of the system.

3.2 Adapting the VoD Service

In the mapping shown in Table 2, we present the contextual adaptation specification between the *VoD* server and *Client*. Associations between messages of both components are established in the following way: (a) under the initial (*Guest*) context, action *play* is always mapped to *preview*, *record* is mapped to *preview* as well, since the client has no access rights for recording the movies. Action *switch* is mapped to *subscribe*, and a context change is triggered in such a way that we arrive to the (b) *Subscriber* context, where the correspondences between actions remain the same, except for *play*, which is mapped to *view*, *record*, which is mapped to *download*, and *switch* to *unsubscribe*. This last correspondence takes us back to the *initial* context. Within this context, we only define the correspondences for *play*, *record* and *switch*, since making use of the override operator, we can leave the rest of the correspondences unchanged, using the ones defined within the *initial* context. In such a way, access rights can change at run-time (actions *switch* and *(un)subscribe*), depending on the *Client* context (*Guest*, *Subscriber*), and accordingly it dynamically changes message translation. Observe that we have had the correspondence between *quit* and *logout* followed by the context change to *VoD*, representing *VoD* like the *"undefined"* context ({}), in order to indicate that the session finishes.

In contrast with our previous works [4,5], in which mappings were static (a command was always translated into the same sequence of messages), this work represents an improvement on the notation. This new technique intends not only to perform dynamic adaptation by the fact that the system alters its behaviour at run-time, but also because message translation changes during the execution of the system as well, depending on changing conditions of the environment (in this case, depending on client profiles). In Table 2 we can observe the contrast between our previous *soft* mappings, where a specification of the subservices (*preview* is a subservice of *view* and *preview* is a subservice of *download*) had to be supplied along with the mappings. With that approach the conditions for a change between service/subservice were left unspecified. On the other hand, the current *contextual* mappings provide a compact representation of all the information required for adaptation, and supply a true dynamic mechanism for it, specifying the conditions which will determine a contextual change.

3.3 Semantics and Adaptor Construction

In this section, we analyze the formal semantics of adaptor specifications. First we will define the semantics for both contextual environments and mappings, and later on, we will use these definitions in order to obtain a contextual adaptor specification from which the actual adaptor will be generated.

We define the semantics of an environment as:

$$[\,] : \Gamma \longrightarrow L_{\perp} \longrightarrow \Gamma$$

Contextual mapping : correspondence of actions
$\mu = \lambda\gamma.\{hello! \rightarrow login?;$ $play! \rightarrow preview?;$ $record! \rightarrow preview?;$ $switch! \rightarrow subscribe?,$ $\{play! \rightarrow view?;$ $record! \rightarrow download?;$ $switch! \rightarrow unsubscribe?, \gamma\} \rhd \gamma;$ $quit! \rightarrow logout?, \{\}\}$
Soft mapping : correspondence of actions
$S = \{hello! \rightarrow login?;$ $play! \rightarrow view?;$ (*where* $preview? < view?,$ $record! \rightarrow download?;$ $preview? < download?$) $quit! \rightarrow logout?\}$

Table 2
Comparison between *soft* mappings and the current *contextual* ones for *VoD*.

$$[\gamma]a = \begin{cases} \gamma & \text{if } \gamma(a) \in 2^R \\ \gamma' & \text{if } \gamma(a) = (x, \gamma') \in 2^R \times \Gamma \\ u_\perp & \text{if } \gamma(a) = \perp \end{cases}$$

where $u_\perp \in \Gamma$ denotes the *undefined* function ($\forall\, a \in L_\perp : u_\perp(a) = \perp$). Extending the semantics for action sequences:

$$[\![\ ]\!] : \Gamma \longrightarrow L_\perp^* \longrightarrow \Gamma$$

$$[\![\, \gamma \,]\!] a.A = \begin{cases} [\gamma]a & \text{if } A = \varepsilon \\ [\![\, [\gamma]a \,]\!] A & \text{if } A \neq \varepsilon \end{cases}$$

The semantics of $[\![\ ]\!]$ defines how an environment evolves when it is applied to a sequence of actions. Considering the example of Section 3.1, where we have the following environment:

$$\begin{aligned} \gamma = \{\ & a \rightarrow 1; \\ & b \rightarrow 2, \{\ a \rightarrow 3;\ \ b \rightarrow 4\ \}; \\ & c \rightarrow 5\ \} \end{aligned}$$

we can illustrate the semantics definition of environment. We apply the semantics of the environment γ on the sequence of actions "$a \cdot b$". First, there is no context change (when applying on a), so we get the same environment. Then, we apply the environment on b, getting a different environment (context change, since b is

mapped to 2, and it is produced a context alteration, a being mapped to 3 and b to 4 in the new context).

We define the semantics of a mapping as:

$$S : M \longrightarrow \Gamma \longrightarrow L_\perp \longrightarrow \Gamma$$

$$S\,\mu\,\gamma\,a = [\mu(\gamma)]a$$

Just as we did before, if we consider an extension of the definition for sequences of actions, we obtain:

$$\boldsymbol{S} : M \longrightarrow \Gamma \longrightarrow L_\perp^* \longrightarrow \Gamma$$

$$\boldsymbol{S}\,\mu\,\gamma\,a.A = \begin{cases} S\,\mu\,\gamma\,a & \textit{if } A = \varepsilon \\ \boldsymbol{S}\,\mu(S\,\mu\,\gamma\,a)A & \textit{if } A \neq \varepsilon \end{cases}$$

The semantics $\boldsymbol{S}$ defines how a mapping evolves when it is applied to a sequence of actions. Following the exemplification of semantics of an environment, we could act of similar way in case of the semantics of a mapping. In order to specify a context-aware adaptor making use of the available information in our contextual mappings, we build a graph, where each of its nodes corresponds to the different contexts contained in the mapping. As we can observe in Figure 1, transitions will be determined by those messages which trigger a contextual change (those of the form $\gamma(a) = (x, \gamma')$). Since the override ($\rhd$) operator allows the generation of an arbitrary number of syntactically different environments, such as $\gamma \rhd \gamma', \gamma \rhd \gamma' \rhd \gamma'', \gamma \rhd \gamma' \rhd \gamma'' \rhd \gamma ...$, we use a transformational definition of $\rhd$ in the construction of the graph:

$$\gamma \rhd \gamma' = \{a \to X \mid a \to X \in \gamma\} \cup \{a \to X \mid a \to X \in \gamma' and \gamma(a) = \perp\}$$

The graph $G = (N, T)$ corresponding to a dynamic mapping μ is built as the result of a finite number of iterations on the algorithm below. We will denote by $G_i = (N_i, T_i)$ the (partial) graph obtained in the *i-th* iteration, where N_i represents the set of nodes and T_i the set of transitions.

In order to be able of understanding the graph construction, we will denote by $(\gamma, < a, x >, \gamma')$ the transition from γ to γ', when $a \in L_\perp, x \in R$ are actions, and γ, $\gamma' \in \Gamma$ are contexts, satisfying $\mu\,\gamma\,a = (x, \gamma')$.

The algorithm will start with the initial graph $G_0 = (\mu\, u_0, \varnothing)$ where u_0 represents an *"arbitrary"* context (note that we use an *"arbitrary"* context, and not the *"undefined"* one).

Given the partial graph constructed in the iteration k, $G_k = (N_k, T_k)$, to build the graph G_{k+1}, let's consider any sequence of transitions $t_0, ..., t_k \in T_k$, such that $t_i = (\gamma_i, < a_i, x_i >, \gamma_{i+1})$, for $i = 0, ..., k-1$, and $\gamma_0 = \mu\, u_0$. Then, we build a new transition set as follows:

$$(1) \qquad T_{k+1} = T_k \cup \{(\gamma_k, < a, x >, \gamma) : \exists\, j (0 \leq j \leq k).\, \mu\,\gamma_k\, a_0 ... a_{j-1}\, a = (x, \gamma)\}$$

and a new node set as:

$$N_{k+1} = N_k \cup \{\gamma : (\gamma_k, < a, x >, \gamma) \in T_{k+1}\}$$

where γ will be a new context or one of the existing contexts.

The algorithm finishes when the set of nodes N_{k+1} coincides with the previous one N_k. Observe that the construcion of the graph always terminates, since the number of different environments that can be generated is finite. Indeed, each environment contains at most one association for each action in L (by definition of $\triangleright$), and the number of actions of L that occur in μ is finite (by definition of mapping and environment).

Note that for $k = 0$, no transition exist in T_0, and then:

$$T_1 = T_0 \cup \{(\mu\, u_0, < a, x >, \gamma) : j = 0.\, \mu\, \gamma_0\, a = (x, \gamma)\}$$

It is worth explaining the condition to add new transitions in (1). The idea behind of considering $\mu\, \gamma_k\, a_0 ... a_{j-1}\, a, (j = 0...k)$, is maintaining information about all the actions which triggered any previous context change.

Considering the mapping in our example:

$$\begin{aligned}\mu = \lambda\gamma.\{\ & \mathit{hello}! \rightarrow \mathit{login}?;\\ & \mathit{play}! \rightarrow \mathit{preview}?;\\ & \mathit{record}! \rightarrow \mathit{preview}?;\\ & \mathit{switch}! \rightarrow \mathit{subscribe}?, \{\ \mathit{play}! \rightarrow \mathit{view}?;\\ & \qquad\qquad\qquad\qquad\quad \mathit{record}! \rightarrow \mathit{download}?;\\ & \qquad\qquad\qquad\qquad\quad \mathit{switch}! \rightarrow \mathit{unsubscribe}?, \gamma\ \} \triangleright \gamma;\\ & \mathit{quit}! \rightarrow \mathit{logout}?, \{\}\ \}\end{aligned}$$

Let's consider an *"initial"* state u_0, which represents an *"arbitrary"* context. Therefore, initially we have:

$$N_0 = \{\mathit{Guest}\}$$
$$T_0 = \{\}$$

where $\mathit{Guest} = \mu\, u_0$ corresponds to the context:

$$\begin{aligned}\mathit{Guest}\{\ & \mathit{hello}! \rightarrow \mathit{login}?;\\ & \mathit{play}! \rightarrow \mathit{preview}?;\\ & \mathit{record}! \rightarrow \mathit{preview}?;\\ & \mathit{switch}! \rightarrow \mathit{subscribe}?, \{\ \mathit{play}! \rightarrow \mathit{view}?;\\ & \qquad\qquad\qquad\qquad\quad \mathit{record}! \rightarrow \mathit{download}?;\\ & \qquad\qquad\qquad\qquad\quad \mathit{switch}! \rightarrow \mathit{unsubscribe}?, u_0\ \} \triangleright u_0;\\ & \mathit{quit}! \rightarrow \mathit{logout}?, \{\}\ \}\end{aligned}$$

Now, we look for all the pairs of actions $a \in L_\perp, x \in R$, and new contexts $\gamma \in \Gamma$, such that $\mathit{Guest}(a) = (x, \gamma)$ (following the algorithm for $k = 0$), and we get two pairs of actions and two new contexts:

$$\mu\, \mathit{Guest}\ \mathit{quit}! = (\mathit{logout}?, \mathit{VoD})$$

$\mu\, Guest\ \ switch! = (subscribe?, Subscriber)$

where:

$VoD = \{\}$ (the *"undefined"* context)

$Subscriber = \{\ play! \rightarrow view?;$
$\qquad\qquad\quad record! \rightarrow download?;$
$\qquad\qquad\quad switch! \rightarrow unsubscribe?, Guest\ \} \rhd Guest$

Thus, we have the new partial graph given by:

$N_1 = \{Guest, VoD, Subscriber\}$

$T_1 = \{\ (Guest, < quit, logout >, VoD),$
$\qquad (Guest, < switch, subscribe >, Subscriber)\ \}$

In the next iteration ($k = 1$), we have two transition (singleton) sequences:

$(Guest, < quit, logout >, VoD)$

$(Guest, < switch, subscribe >, Subscriber)$

but the first one does not generate any new transition because *VoD* is the *"undefined"* context. However, following the definition of the new transition set T_1, we have that:

$\mu\, Subscriber\ \ quit! = (logout?, VoD)$ (for $j = 0$ in (1))

$\mu\, Subscriber\ \ switch!\ \ switch! = (unsubscribe?, Guest)$ (for $j = 1$ in (1))

which produces the new graph:

$N_2 = \{Guest, VoD, Subscriber\}$

$T_2 = \{\ (Guest, < quit, logout >, VoD),$
$\qquad (Guest, < switch, subscribe >, Subscriber),$
$\qquad (Subscriber, < quit, logout >, VoD),$
$\qquad (Subscriber, < switch, unsubscribe >, Guest)\}$

Now, there is no any context which triggers a new one, so the graph construction finishes.

Following the construction mechanism for our example, the graph we obtain for the *VoD* client/server interaction contextual adaptor is shown in Figure 1:

Once we have the specification of the contextual adaptor, this will be constructed by:

1. Deriving an adaptor for each of the contexts γ_i in N using the algorithm described in [4]. Note that the actions which belong to a correspondence $\gamma(a) = (x, \gamma')$ are not incorporated to the adaptor.

2. Assembling each of the adaptors built by making use of the transitions already defined. So, the contextual adaptor will decide the adaptor currently being used

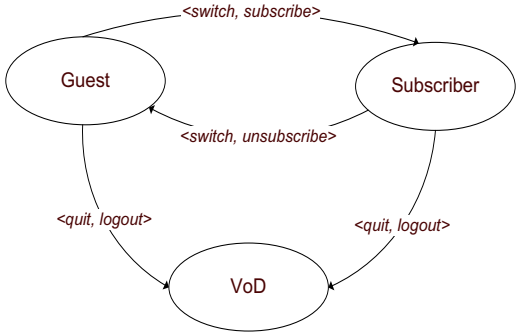

Fig. 1. Graph adaptor specification for the *VoD* service.

based on the messages received in the interaction. These transitions are inserted by means of the actions which belong to a $\gamma(a) = (x, \gamma')$ correspondence (actions of the adaptors specified in Table 3 in bold). The action is inserted along with the reference of the adaptor which corresponds to the target state of the transition specified in the graph.

It is worth noting that is no trivial to get directly an adaptor in a process algebra (like CCS). In our proposal, we make use of contexts to determine the translation between the messages of the components being adapted, depending on the changing conditions of the environment while it is running.

$$
\begin{aligned}
A &= hello?login!A_{Guest} \\
A_{Guest} &= play?preview!A_{Guest} \\
&\quad + record?preview!A_{Guest} \\
&\quad + \boldsymbol{switch?subscribe!A_{Subs}} \\
&\quad + \boldsymbol{quit?logout!A_{VoD}} \\
A_{Subs} &= play?view!A_{Subs} \\
&\quad + record?download!A_{Subs} \\
&\quad + \boldsymbol{switch?unsubscribe!A_{Guest}} \\
&\quad + \boldsymbol{quit?logout!A_{VoD}} \\
A_{VoD} &= \oslash
\end{aligned}
$$

Table 3
VoD contextual adaptor specification.

Once we have obtained this graph, making use of the semantic rules we have defined, we will be able to demonstrate security properties from the server's point of view, such as that no *download* action will take place if a *subscribe* action has not been performed before (this action will trigger the transition to the *Subscriber* context), or other kind of properties such as deadlock freedom and termination. In Figure 2 (shown below) we can observe the *VoD* client/server interaction, where the

contextual adaptor alternates the use of the A_{Guest} and A_{Subs} adaptors for message translation with each subsequent *switch* message it receives on behalf of the client. The adaptor finishes when it receives a *quit* command in any case.

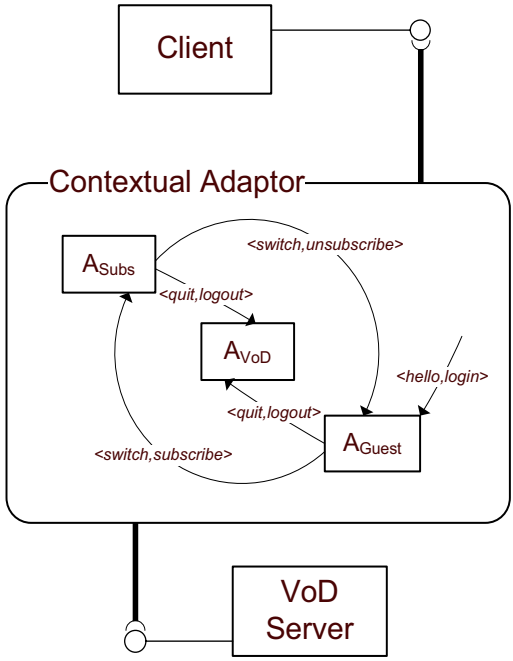

Fig. 2. *VoD* client/server interaction.

From the client's point of view, we want to know if the server is going to allow us to perform the actions we are granted once we have subscribed to the service, because we can arrive to situations in which the final adaptor we have may not enable the actions we want to perform.

4 Conclusions and Further Work

We have presented throughout this paper a description of a formal notation for contextual component adaptation. The purpose of this new technique is to obtain dynamic mappings between the interfaces of the components being adapted, through contextual environments that define flexible adaptation policies. In previous works, this idea was presented, although from the point of view of *subservicing*. However, this approach lacked the flexibility of the current work, where we intend to overcome some of the previous limitations, making a significant advance to find a solution for issues like dynamic access rights (user privileges) by means of contextual adaptation, as we can see in our example.

We have defined the syntax and semantics of contextual environments and mappings, obtaining a high degree of expressiveness in order to describe contextual adaptors, enabling a changing message translation between the components depending on the conditions of the system and the course of component interaction.

In order to exemplify our proposal we have presented a case study in this paper relative to a *Video-on-Demand* (VoD) system with different client profiles (used as contextual environments). Having a contextual mapping available, we have presented an algorithm to build the specification of the contextual adaptor based on a graph representing the contexts present in the mapping, and the transitions between them. Finally, we have sketched the procedure to deploy an actual contextual adaptor based on the aforementioned specification and the algorithm described in [4].

Although this notation represents a significant leap for the expressiveness of the adaptor, it still has certain limitations. It is worth mentioning that this proposal constitutes a modular and on-going approach to the specification of the required adaptation between two software components. Thus, an interesting extension is to consider adaptation between manifold interacting components.

In addition, the possibility that the services may not be available at some point during the execution will have to be considered in future work. Finally, component interoperability at the semantic level, which is a complex task to tackle, has to be studied in depth in order to support the automatic generation of consistent mappings between component interfaces.

It would also be interesting to introduce security policies for the dynamic adaptation which we have proposed in this work. The adaptor must guarantee the safe interaction of the adapted components (i.e. verification of properties), making sure that they will never deadlock during an interaction session.

References

[1] Allen, R. and D. Garlan, *A Formal Basis for Achitectural Connection*, In ACM Trans. on Software Engineering and Methodology **6** (1997), pp. 213–249.

[2] Becker, S., A. Brogi, I. Gorton, S. Overhage, A. Romanovsky and T. M., *Towards an Engineering Approach to Component Adaptation*, in: *Dagstuhl Seminar 04511: Architecting Systems with Trustworthy Components*, Springer-Verlag, Lecture Notes in Computer Science **3938**, 2006.

[3] Bergel, A., S. Ducasse and O. Nierstrasz, *Analyzing Module Diversity*, Journal of Universal Computer Science **11** (2005), pp. 1613–1644.

[4] Bracciali, A., A. Brogi and C. Canal, *A Formal Approach to Component Adaptation*, Journal of Systems and Software, Special Issue on Automated Component-Based Software Engineering **74** (2005), pp. 45–54.

[5] Brogi, A., C. Canal and E. Pimentel, *Component adaptation through flexible subservicing*, Science of Computer Programming, Elsevier (in press) (2006).

[6] Canal, C., L. Fuentes, E. Pimentel, J. Troya and A. Vallecillo, *Adding Roles to CORBA Objects*, IEEE Transactions on Software Engineering **29** (2003), pp. 242–260.

[7] Canal, C., J. Murillo and P. Poizat, *Software adaptation*, in: *Special Issue on the 1st International Workshop on Coordination and Adaptation of Software Entities (WCAT'04)*, L'Objet **12**, 2006, pp. 9–31.

[8] Canal, C., P. Poizat and G. Salaün, *Synchronizing Behavioural mismatch in software composition*, in: *Proc. of FMOODS'06*, Springer-Verlag, 2006.

[9] Honda, K., V. Vasconcelos and M. Kubo, *Language primitives and type disciplines for structured communication-based programming*, in: *European Symposium on Programming (ESOP'98)*, Springer-Verlag, Lecture Notes in Computer Science **1381**, 1998, pp. 122–138.

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

[11] Magee, J., J. Kramer and D. Giannakopoulou, *Behaviour analysis of software architectures*, In Software Architecture (1999), pp. 35–49.

[12] Nierstrasz, O. and T. Meijler, *Research Directions in Software Composition*, ACM Computing Surveys **27** (1995), pp. 262–264.

[13] Wagner, F., R. Schmuki, T. Wagner and P. Wolstenholme, *Modeling Software with Finite State Machines*, A Practical Approach, CRC Press (2006).

[14] Yellin, D. and R. Strom, *Protocol specifications and components adaptors*, In ACM Transactions on Programming Languages and Systems **19** (1997), pp. 292–333.