Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 97 (2004) 47–65
www.elsevier.com/locate/entcs

On the specification of software adaptation

Antonio Brogi[1]

Dipartimento di Informatica, Università di Pisa, Italy

Carlos Canal[2] Ernesto Pimentel[3]

Dpto. de Lenguajes y Ciencias de la Computación, Univ. de Málaga, Spain

Abstract

The problem of adapting heterogeneous software components that present mismatching interaction behaviour is one of the crucial problems in Component-Based Software Engineering. The aim of this paper is to contribute to setting a theoretical foundation for software adaptation. A formal analysis of adaptor specifications is presented, and their usage to feature different forms of flexible adaptations is illustrated.

Keywords: Software composition, formal methods, process algebras.

1 Introduction

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering [4,7]. 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 [3]. Available component-oriented platforms feature Interface Description Languages (IDLs) to address software interoperability at the signature level. IDLs are a sort of

[1] Email: brogi@di.unipi.it
[2] Email: canal@lcc.uma.es
[3] Email: ernesto@lcc.uma.es

1571-0661
doi:10.1016/j.entcs.2004.04.031

lingua franca for specifying the functionalities offered by heterogeneous components that were developed in different languages. While IDL interfaces allow to overcome signature mismatches between components, there is no guarantee that the components will suitably interoperate as mismatches may also occur at the protocol level, because of differences in the interaction behaviour of the components involved [8].

In our previous work [1], we have developed a formal methodology for component adaptation that supports the successful interoperation of heterogeneous components presenting mismatching interaction behaviour. 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 are simply expressed by a set of correspondences between actions 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 methodology has proven to succeed in a number of diverse situations [1], where a suitable adaptor is generated to support the successful interoperation of heterogeneous components presenting mismatching interaction behaviour. The separation of adaptor specification and adaptor derivation permits the automation of the error-prone, time-consuming task of constructing a detailed implementation of a correct adaptor, thus notably simplifying the task of the (human) software developer.

One of the distinguishing features of the methodology is the simplicity of the notation employed to express adaptor specifications. Indeed the desired adaptation is simply expressed by defining a set of (possibly non-deterministic) correspondences between the actions of the two components. While adaptor specifications have been thoroughly employed in [1] to address various examples of adaptation, a formal and precise characterisation of these specifications had not been developed.

The aim of this paper is precisely to set a theoretical foundation for software adaptation. In particular, after presenting a simple motivating example to illustrate the adaptation methodology (Sect. 2), we will focus on adaptor specifications and start by presenting their precise syntax (Sect. 3). We

shall then analyse the formal semantics of adaptor specifications (Sect. 4), and show how a specification defines a set of processes that describe the interaction behaviour of the adaptor components capable of featuring the desired adaptation. We will also show that the defined semantics induces a partial order and an equivalence relation over adaptor specifications, which can be used to reason and to prove useful properties about them. We will then move (Sect. 5) to analyse how the process of adaptation can be formally described as a transformation over adaptor specifications, and how this helps in understanding the meaning of the overall adaptation process. A more flexible form of soft adaptation will be then formally presented (Sect. 6), where the notion of sub-servicing is employed to weaken the initial adaptor specification when the latter cannot be fully realised. The possibility of expressing hard requirements in adaptor specifications will be then illustrated (Sect. 7), and their effect on adaptor generation will be described. Finally some concluding remarks will be drawn (Sect. 8).

We will try to employ simple examples to illustrate the ideas described. While we hope that those examples will provide enough intuition in spite of their simplicity, the interested reader is referred to [1,2] for more significant examples of software adaptation.

Notice also that while in [1] adaptor specifications may include data dependencies, we will focus here only on action correspondences for the sake of simplicity. Correspondingly we will omit input/output signs of actions in the sequel as this notably simplifies the discussion without loss of generality.

2 An example of software adaptation

To provide the context, we first illustrate a simple example of software adaptation. Following [1], we assume that component interfaces include *interaction patterns* that describe the essential aspects of the *finite* behaviour that a component may (repeatedly) show to the external environment. Syntactically, these patterns are terms of a process algebra (a subset of π-calculus in [1]).

Consider for instance a simple server P that offers a query-answering service. Namely, the server waits for receiving a query and then returns an answer for such a query. The interaction protocol followed by P can be expressed by the interaction pattern:

$$query?().result!().0$$

Consider now a client Q that issues a query and waits for an answer, but it may also decide to stop waiting and abort the request. Suppose that the

behaviour of Q is expressed by the interaction pattern:

$$request!().(reply?().0 + tau.abort!().0)$$

It is worth observing that the mismatch between the above two components is not limited to signature differences (viz., the different names of actions employed), but it also involves behavioural differences.

The objective of software adaptation is to deploy a software component, called *adaptor*, capable of acting as a component-in-the-middle between P and Q and capable of supporting their successful interoperation. A concrete adaptor will be automatically generated starting from the interfaces of the components and from a specification of the adaptor itself. Such a specification simply consists of rules establishing correspondences between actions of the two components. The natural specification of the adaptor for the example at hand is:

$$\left\{ \begin{array}{l} query \Diamond request; \\ result \Diamond reply; \\ result \Diamond abort \end{array} \right\}$$

which establishes a correspondence between actions *query* and *request*, and which simply states (as we shall see later) that action *result* may nondeterministically correspond to either *reply* or *abort*, depending on the evolution of the client Q.

Given an adaptor specification, a fully automated procedure [1] returns (if possible) an adaptor component that satisfies the specification and that lets the two component successfully interoperate. For this example, the process will return the adaptor:

$$request?().query!().result?().(reply!().0 + abort?().0)$$

3 Syntax of adaptor specifications

An adaptor specification is a set of rules of the form:

$$\alpha_1, \ldots, \alpha_m \Diamond \beta_1, \ldots, \beta_n$$

where α_i and β_j are input or output actions to be (possibly) performed by the adaptor component. By convention, actions on the left side of rules refer to one of the components to be adapted, while actions on the right side of rules refer to the other component. As we mentioned at the end of the introduction,

we are omitting input and output signs of actions. For instance the rule:

$$a \Diamond c$$

is used to specify that whenever the adaptor will perform an action a, it will have to eventually perform a corresponding action c, or vice-versa. Similarly, the rule:

$$a, b \Diamond c$$

specifies that whenever the adaptor will perform an action a (or b), it will have to eventually perform an action b (a respectively) as well as an action c.

The adaptation needed to let two parties interoperate may have to cope with asymmetries, typically when an action of one component does not have a corresponding action in the other component. This situation is naturally expressed by means of rules having an empty side. For instance the rule:

$$a \Diamond$$

specifies that while the adaptor may need to perform an action a to match an action of one of the components to be adapted, there is no corresponding action to be performed w.r.t. the other component.

Notice that the above rules allow an arbitrary interleaving of different occurrences of the actions specified in a correspondence rule. For instance — as we shall see formally later on — the rule:

$$a \Diamond b$$

will be satisfied both by the adaptor $a.b.a.b.0$ and by the adaptor $a.a.b.b.0$.

The syntax of adaptor specifications hence features a second operator $\Diamond\!\!\!\!\!\!\Diamond$ to express tighter correspondences among (sets of) actions in a rule. Namely the operator $\Diamond\!\!\!\!\!\!\Diamond$ does not allow the interleaving of different occurrences of actions from a correspondence rule. For instance the rule:

$$a \Diamond\!\!\!\!\!\!\Diamond b$$

will be satisfied by the adaptor $a.b.a.b.0$ but not by the adaptor $a.a.b.b.0$.

An adaptor specification is hence a (finite) set of rules, separated by ";". Notice that the syntax for rules allows nondeterminism in the specification of

actions correspondences. For instance, a specification such as:

$$\left\{ \begin{array}{l} a \diamondsuit b\ ; \\ a \diamondsuit c, d\ ; \\ a \diamondsuit \end{array} \right\}$$

states that if the adaptor performs an action a, it may either perform action b, or the pair of actions c and d, or even none of them.

4 Semantics of adaptor specifications

An adaptor specification defines the properties that the behaviour of an adaptor component must satisfy. Each rule in a specification can be (automatically) translated into a property and expressed as a process algebra term. For instance the specification:

$$\left\{ \begin{array}{l} a \diamondsuit b\ ; \\ \diamondsuit c \end{array} \right\}$$

translates into the two properties (one per rule):

$$R_1 = a.(b.0|R_1) + b.(a.0|R_1) + \tau.0$$
$$R_2 = c.(0|R_2) + \tau.0$$

Intuitively speaking, property R_1 states that if the adaptor will perform an action a (or b), then it will have to eventually perform an action b (a respectively) — i.e., actions a and b must be performed in pairs, though they may freely interleave. Notice that the number of pairs of a's and b's is not determined, though process R_1 may eventually stop via an internal τ move. Similarly, property R_2 simply states that the adaptor may perform an arbitrary number $n \geq 0$ of times action c.

Rules inhibiting the interleaving of different occurrences of actions are translated accordingly. For instance the rule:

$$d() \diamondsuit\!\!\!\!\diamondsuit e();$$

translates into the property:

$$R_3 = d.(e.R_3) + e.(d.R_3) + \tau.0$$

Notice how R_3 states for instance that if the adaptor will perform action d, then it will have to perform an action e before being allowed to perform another d.

For a given adaptor specification S, we will denote by $\Pi(S)$ the parallel composition of the properties defined by the rules in S. The set of processes defined by an adaptor specification S is then the set of processes which are *simulated* by the process $\Pi(S)$. Since processes are meant to describe intensional behaviour (of both adaptors and properties), synchronizations are not allowed within processes. Formally, the following non-synchronizing semantics of processes is used:

$$\frac{}{a.P \xrightarrow{a} P} \qquad \frac{}{\tau.P \xrightarrow{\tau} P} \qquad \frac{P \xrightarrow{x} P'}{P + Q \xrightarrow{x} P'} \qquad \frac{P \xrightarrow{x} P'}{P|Q \xrightarrow{x} P'|Q}$$

(together with the standard commutativity and associativity axioms for $+$ and $|$). We will denote by $P \xrightarrow{\tau}{}^{*} P'$ the fact that P can evolve into P' with a (finite) number of τ transitions.

We can now formally define the notion of simulation between processes.

Definition 4.1 A process P is *simulated* by Q ($P \preceq Q$) if and only if

(1) If $P \xrightarrow{a} P'$ then $(Q \xrightarrow{a} Q' \ \wedge \ P' \preceq Q')$, and
(2) If $P \equiv 0$ then $(Q \xrightarrow{\tau}{}^{*} Q' \wedge Q' \equiv 0)$.

We will use the above notion of process simulation to characterise the set of processes that satisfy a given adaptor specification S. Notice that, as component interfaces are expressed by finite interaction patterns, we are interested in *finite* processes capable to adapt such patterns. (Indeed, the above notion of simulation well characterises finite adaptors, as a non-terminating process such as $P = a.P$ would otherwise satisfy the specification $a \Diamond b$.)

In other words, if we instanciate Definition 4.1 to the case in which P is finite, we have that P is simulated by Q if and only if for every trace such that $P \xrightarrow{\alpha_1}\xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} P'$ and $P' \equiv 0$, then $Q \xrightarrow{\alpha_1}\xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n}\xrightarrow{\tau}{}^{*} Q'$ and $Q' \equiv 0$.

We can now formally define the set of processes that satisfy an adaptor specification S as the set of processes that are simulated by the process $\Pi(S)$.

Definition 4.2 A process P *satisfies* an adaptor specification S if and only if $P \preceq \Pi(S)$. We will denote by $[\![S]\!]$ the set of all processes that satisfy an adaptor specification S, that is: $[\![S]\!] = \{P \mid P \preceq \Pi(S)\}$.

Notice that in general $[\![S]\!]$ denotes an infinite set of processes. Consider

for instance again the specification S:

$$\left\{ \begin{array}{c} a \diamondsuit b \,; \\ \diamondsuit\, c \end{array} \right\}$$

The set $[\![S]\!]$ will contain all the processes simulated by $\Pi(S) = (R_1|R_2)$, where:

$$R_1 = a.(b.0|R_1) + b.(a.0|R_1) + \tau.0$$
$$R_2 = c.(0|R_2) + \tau.0$$

Namely $[\![S]\!]$ will contain the processes 0, $c.0$, $a.b.0$, $a.b.c.0$, $c.a.c.b.c.0$, as well as $(a.b.0) + (c.b.a.0)$, and so on and so forth. On the other hand, $[\![S]\!]$ will not include for instance processes $a.b.a.0$, $c.b.0$ or $d.0$.

The above denotation $[\![\cdot]\!]$ directly induces an ordering on adaptor specifications.

Definition 4.3 Given two adaptor specifications S_1 and S_2, we put:

$$S_1 \leq S_2 \;\text{ if and only if }\; [\![S_1]\!] \subseteq [\![S_2]\!].$$

Namely, $S_1 \leq S_2$ means that the specification S_1 admits less processes than S_2. It is easy to see, for instance, that:

$$S_1 \;=\; \{a \diamondsuit\} \quad \leq \quad S_2 \;=\; \{a \diamondsuit \,;\, \diamondsuit\, b\}$$

while $S_2 \not\leq S_1$ since, for instance, $a.b.0 \in [\![S_2]\!] \setminus [\![S_1]\!]$ as S_1 does not allow performing b.

The ordering $\leq$ is (trivially) reflexive and transitive, and induces the following equivalence relation on adaptor specifications:

$$S_1 \equiv S_2 \;\text{ if and only if } (S_1 \leq S_2 \text{ and } S_2 \leq S_1).$$

It is also easy to see that the empty specification $\emptyset$ is the least element in the $\leq$-ordering, as $[\![\emptyset]\!]$ is the empty set of processes. On the other hand, the

specification:

$$\left\{\begin{array}{l} l_1 \Diamond \ ; \\ \quad \vdots \\ l_m \Diamond \ ; \\ \Diamond\ r_1\ ; \\ \quad \vdots \\ \Diamond\ r_n\ ; \end{array}\right\}$$

(where $V = \{l_1, \ldots, l_m, r_1, \ldots, r_n\}$ is the vocabulary of all actions considered) is the largest [4] specification in the $\leq$-ordering.

It is now worth stating some properties of adaptor specifications, which help to understand their meaning and usage. The first property below (Proposition 4.4) shows that extending a specification with a new rule actually corresponds to enlarging the set of adaptors specified. The second property formalises the expected relation between the $\Diamond\!\!\!\!\Diamond$ and $\Diamond$ operators, in the sense that the former is more constraining than the latter. Finally the third property shows that the union of specifications preserves the $\leq$-ordering.

Proposition 4.4 *Let S,S' be adaptor specifications, r be a correspondence rule, and $\alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_n$ actions. Then:*

(1) $S \leq S \cup \{r\}$
(2) $\{\alpha_1, \ldots, \alpha_m \Diamond\!\!\!\!\Diamond \beta_1, \ldots, \beta_n\} \leq \{\alpha_1, \ldots, \alpha_m \Diamond \beta_1, \ldots, \beta_n\}$
(3) $S' \leq S$ *implies* $S \cup S' \leq S$

As a direct consequence of this proposition, we can also prove other interesting properties which also help us to understand the meaning of adaptors. The first property below (Corollary 4.5) shows that extending a specification S with a subsumed specification does not alter the meaning of S. The second property states that a specification that subsumes two other specifications S_1 and S_2, also subsumes their union $S_1 \cup S_2$. Finally the third property establishes that the least upper bound (lub) of two specifications coincides with their union

Corollary 4.5 *Let S, S_1, and S_2 be adaptor specifications. Then:*

(1) $S_1 \leq S$ *iff* $S_1 \cup S \leq S$
(2) $S_1 \leq S$ *and* $S_2 \leq S$ *imply* $S_1 \cup S_2 \leq S$

[4] Note that there are infinite specifications that are equivalent to this largest specification. They can be obtained for instance by arbitrarily adding rules of the form: $l_i, l_i, \ldots, l_i \Diamond$, to such a specification.

(3) $lub(S_1, S2) = S_1 \cup S_2$

It is worth formally introducing the notions of *extension* and *reduction* over specifications, as they will be often referred to in the sequel.

Definition 4.6 Let S_1, S_2 be adaptor specifications. We say that S_2 is an *extension* of S_1 if and only if $S_2 = S_1 \cup T$, for some adaptor specification T. We also say that S_1 is a *reduction* of S_2 whenever S_2 is an extension of S_1.

Obviously, by Proposition 4.4, if S_1 is a reduction of S_2 then $S_1 \leq S_2$. Notice however that the converse is not true in general. For instance

$$\{a \langle\!\rangle b\} \leq \{a \Diamond b\}$$

while the first specification is not a reduction of the second one.

5 Adaptor specifications as contract agreements

Adaptor specifications can be employed to specify the desired adaptation between two components that present mismatching interaction behaviour. Given an adaptor specification and the interfaces of the components to be adapted, the automatic procedure described in [1] derives (if possible) a concrete adaptor by exhaustively trying to build a component which satisfies the given specification while letting the components successfully interoperate. While the ultimate result of the process of software adaptation is a concrete adaptor component (if any), in many situations it is more convenient to present such a result in the form of an adaptor specification.

Consider for instance a typical asymmetric scenario where a client component wishes to use some of the services offered by a server. (For instance a client wishing to access a remote system via the network, or a mobile client getting into the vicinity of a stationary server.) The client will ask for the server interface, and then submit its service request in the form of an adaptor specification (together with its own interface). The server will run the adaptor derivation procedure to determine whether a suitable adaptor can be generated to satisfy the client request. If the client request can be satisfied, the server will notify the client by presenting a (possibly modified) adaptor specification which states the type of adaptation that will be effectively supported. The client will then decide whether to accept the proposed adaptation or not. (Notice that in the latter case the client may decide to continue the trading process by submitting a different adaptor specification.)

Expressing adaptation trading by means of adaptor specification features two main advantages:

- *Efficiency* — Clients and servers exchange light-weighted adaptor specifications rather than component code. Besides affecting the efficiency of communications, this notably simplifies the trading process, when the client has to analyse the adaptation proposed by the server.
- *Non-disclosure* — The server does not have to present the actual adaptor component in its full details, thus communicating only the "what" of the offered adaptation rather than the "how".

Summing up, the communications of this adaptation trading reduce to an adaptor specification S, representing the client request, and to a (possibly modified) adaptor specification C, representing the actual adaptation offered by the server. The specification C is then interpreted as the contract guaranteeing that:

(i) The client will successfully interoperate with the adaptor (viz., the client will not get stuck), and

(ii) all the client actions occurring in C will be effectively executable by the client.

To illustrate the idea, consider a client wishing to access a simple video-on-demand service. The client wishes to perform its *info* and *play* actions to request information on available movies and to view a movie, respectively, and to exploit its *data* action to receive data from the server. The client hence submits the following adaptor specification S (by establishing correspondences with the server actions), and suppose that it receives the following proposed contract C:

$$S = \left\{ \begin{array}{l} info \Diamond search\ ; \\ play \Diamond view, start\ ; \\ data \Diamond stream \end{array} \right\} \qquad C = \left\{ \begin{array}{l} info \Diamond search\ ; \\ data \Diamond stream \end{array} \right\}$$

The straightforward reading of the proposed contract C is that while the server commits to let the client access information on movies, it will not feature the adaptation required to let the client view such movies. (Notice that the server might decide to feature a partial adaptation even if a full adaptation would be feasible, for instance to balance its current workload or for other internal service policies.)

Partial adaptation may simply consist of (as in the example above) removing some correspondence rules from the submitted specification. Notice that in such cases the proposed contract C is a reduction of the submitted request S, and hence $C \leq S$ by virtue of Proposition 4.4. The type of com-

ponent adaptation that we have described so far, given a proposed adaptor specification S:

- either it yields a (possibly partial) adaptation $C \leq S$,
- or it fails when no partial adaptation is possible.

The sole possibility of removing some rules from the initial specification obviously limits the success possibilities of yielding a (partial) adaptation. Indeed there are many situations in which more flexible ways of weakening the initial specification may lead to deploying a suitable partial adaptor, as we shall discuss in the next section.

6 Soft adaptation

The methodology for software adaptation described in [1] has been recently extended in [2] to feature forms of *soft* adaptation. One of the key notions introduced in [2] is the notion of *sub-service*. Intuitively speaking, a sub-service is a kind of surrogate of a service, which features only a limited part of such service. For instance, in the above video-on-demand service, offering a clip preview of a movie can be considered a typical sub-service of offering the whole movie.

Formally, sub-services are specified by defining a partial order $\sqsubset$ over the actions of a component. For instance, continuing with the example, the relation:

$$preview \quad \sqsubset \quad view$$

states that *preview* is a sub-service of *view* in the video-on-demand server (or, equivalently, *view* is a super-service of *preview*). It is important to observe that adding sub-service declarations to component interfaces paves the way for more flexible forms of adaptation while respecting the separation of concerns advocated by aspect-oriented software development (AOSD). Indeed, sub-service declarations support a flexible configuration of components in view of their (dynamic) adaptation, without having to modify or to make more complex the protocol specification of a component interface.

As one may expect, the introduction of sub-services notably increases the possibilities of successful adaptations, as an initial specification can be suitably weakened by providing (when needed) sub-services in place of the required services. (As in the case of the partial adaptation described in the previous Section, a server may decide to sub-service some of the client requests even if this is not strictly necessary in order to achieve a successful inter-operation of the two components. For instance a server may need to balance its current workload, or handle requests in terms of access rights as discussed in [2].)

A consequence of enabling soft adaptation is that a client that submits an adaptor specification may now receive a rather different proposed contract, in which the server may declare its intention both to feature only some of the services requested and to sub-service some of them. To understand why soft adaptors are not weird answers, we now analyse their meaning in terms of the semantics of adaptor specification described in the previous sections.

Formally, the process of adaptor generation in presence of sub-service declarations can be described as follows.

(i) The initial adaptor specification S is actually interpreted as the specification S^* obtained by expanding S with new correspondence rules that are obtained by replacing services with sub-services in the rules of S in all possible ways. As S^* is an expansion of S, we have that $S \leq S^*$ by virtue of Proposition 4.4.

(ii) The process of adaptor construction generates (if possible) a partial adaptor that satisfies a reduction C of the given specification S^*, and hence it returns a proposed adaptation $C \leq S^*$.

Let us formally introduce the notion of sub-service expansion of an adaptor specification.

Definition 6.1 Let S be an adaptor specification, and let $\sqsubset$ be the sub-service relation over actions in S. The *sub-service expansion* S^* of S is obtained by extending S with the set of all correspondence rules

$$\alpha'_1, \ldots, \alpha'_m \;\circ\; \beta'_1, \ldots, \beta'_n \;;$$

such that

$$\alpha_1, \ldots, \alpha_m \;\circ\; \beta_1, \ldots, \beta_n \;;$$

is a rule of S (where $\circ$ is either $\diamondsuit$ or $\Diamond\!\!\!\!\Diamond$), and where for all i: ($\alpha'_i = \alpha_i$ or $\alpha'_i \sqsubset \alpha_i$) and ($\beta'_i = \beta_i$ or $\beta'_i \sqsubset \beta_i$).

Consider again the simple example of the video-on-demand service, where the adaptor specification initially submitted by the client was:

$$S = \left\{ \begin{array}{l} info \diamondsuit search \;; \\ play \diamondsuit view, start \;; \\ data \diamondsuit stream \end{array} \right\}$$

Suppose that the server interface contains the sub-service declarations

$$preview \sqsubset view \qquad advertise \sqsubset search \qquad advertise \sqsubset preview$$

(where *advertise* consists for instance of projecting an advertisement to invite guests to subscribe). Then the specification S is actually interpreted by the server as:

$$S^* = \left\{ \begin{array}{l} info \Diamond search \ ; \\ info \Diamond advertise \ ; \\ play \Diamond view, start \ ; \\ play \Diamond preview, start \ ; \\ play \Diamond advertise, start \ ; \\ data \Diamond stream \end{array} \right\}$$

where the second, fourth, and fifth rules have been introduced by replacing services with sub-services.

Depending on the component protocols, as well as on the server policy, the server may return different contract proposals, such as:

$$C_1 = \left\{ \begin{array}{l} info \Diamond search \ ; \\ play \Diamond preview, start \ ; \\ data \Diamond stream \end{array} \right\} \quad or \quad C_2 = \left\{ \begin{array}{l} info \Diamond search \ ; \\ play \Diamond preview, start \ ; \\ play \Diamond advertise, start \ ; \\ data \Diamond stream \end{array} \right\}$$

where C_2 indicates that some *play* requests will be adapted into previews while others into advertisements. Notice that the server may actually return *any* partial adaptor for S, including for instance:

$$C_3 = \left\{ \begin{array}{l} play \Diamond advertise, start \ ; \\ data \Diamond stream \end{array} \right\}$$

where the first correspondence rule of S has been removed altogether.

As we have already pointed out, $S \leq S^*$ and $C_i \leq S^*$ for all possible contract proposals C_i returned. However, the interesting question from the point of view of the client is what is the relation between the received contract proposal and the initially proposed specification. The answer is that every contract proposal is a reduction of the initial specification where some services have been possibly sub-serviced, as formalised by the following proposition.

Proposition 6.2 *Let S be an adaptor specification, and let S^* be the sub-service expansion of S. Let σ be a name substitution such that if T is an*

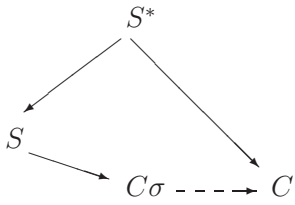

Fig. 1. A graphical view of the result of Proposition 6.2. Solid arrows denote reductions, while the dashed arrow denotes sub-servicing.

adaptor specification then $T\sigma$ is obtained from T by replacing some service name occurrences[5] *in T with a corresponding super-service. Then for each reduction C of S^* there exists a name substitution σ such that $C\sigma$ is a reduction of S.*

For instance, continuing with the example, we have that for

$$\sigma_1 = \{preview \mapsto view\} \text{ and } \sigma_2 = \{preview \mapsto view, advertise \mapsto view\} :$$

$$C_1\sigma_1 \;=\; C_2\sigma_2 \;=\; \left\{ \begin{array}{l} info \Diamond search\,; \\ play \Diamond view, start\,; \\ data \Diamond stream \end{array} \right\} \;=\; S$$

while for $\sigma_3 = \{advertise \mapsto view\}$

$$C_3\sigma_3 = \left\{ \begin{array}{l} play \Diamond view, start\,; \\ data \Diamond stream \end{array} \right\} \;\leq\; S$$

7 Hard requirements

We have seen that adaptor derivation can be described as a transformation over adaptor specifications. Soft adaptation may generate a soft adaptor that does not strictly satisfy the initial adaptor specification. Namely the derived adaptor is described by a specification which is a reduction of the initial specification where some services have been possibly sub-serviced.

[5] Notice that name substitutions must be defined on name occurrences (rather than on names) as sub-servicing may be non-uniform in general. For instance if $x' \sqsubset x$ and $S = \{a\Diamond x;\ b\Diamond x;\}$, then $C = \{a\Diamond x;\ b\Diamond x';\}$ is a reduction of S^* where only the b request for x will be sub-serviced with x'.

On the other hand, while adaptor derivation is free to possibly revise any correspondence given in an initial specification, the proposer of such a specification does not have means to indicate whether there are parts of the specification that are to be interpreted as *hard requirements* which must be satisfied by the adaptor to be generated. The capability of expressing hard requirements in adaptor specifications is obviously very important to drive (and speed-up) the process of adaptation trading.

We therefore extend the syntax of adaptor specifications to allow expressing hard requirements by introducing solid versions ⧫ and ⧓ of the rule correspondence operators ◊ and ⟡. Intuitively speaking, a correspondence rule

$$\alpha_1, \ldots, \alpha_m \blacklozenge \beta_1, \ldots, \beta_n \ ;$$

in a specification S states that such rule should be contained *verbatim* in the proposed contract that will describe the generated adaptor. In other words, such a correspondence should neither be omitted nor sub-serviced during the adaptor generation process.

To illustrate the idea of hard requirements, consider again the video-on-demand example and suppose that the client submits the specification:

$$S = \left\{ \begin{array}{l} info \blacklozenge search \ ; \\ play \Diamond view, start \ ; \\ data \Diamond stream \end{array} \right\}$$

The intended meaning of S is that while the client may consider accepting some sub-servicing for the *view* service, she will not accept adaptations that will not allow her to access the information on available movies.

It is worth noting that the treatment of hard requirements can be smoothly included in the process of adaptor generation described in the previous section.

(i) The initial specification S is interpreted (as before) as the sub-service expansion S^* of S. Notice that hard rules in S are now transformed into their non-hard equivalent (viz., solid operators are turned into their corresponding non-solid version), while the new rules generated by sub-servicing replacements are obtained by expanding only those rules of S that do not represent hard requirements.

(ii) The process of adaptor construction generates (if possible) a partial adaptor that satisfies a reduction C of the given specification S^*. The only difference is that the proposed reduction C of S^* must now include all the hard requirements that were present in S.

Formally, let $S = S_h \cup S_{nh}$ where S_h and S_{nh} denote respectively the set of hard and non-hard requirements in S. Then $S^* = S'_h \cup S_{nh} \cup E$, where S'_h is the non-solid version of S_h and E is the set of rules added to S by the sub-service expansion of S_{nh}. The proposed contract must be then of the form $C = S'_h \cup C_{S_{nh}} \cup C_E$ where $C_{S_{nh}}$ is a reduction of S_{nh} and where C_E is a reduction of E. In other words, the $*$ operator can be extended to hard requirements specifications as follows:

$$S^* = S'_h \cup S^*_{nh}$$

where S^*_{nh} is defined as was explained in previous section.

It is worth noting that Proposition 6.2 continues to hold in presence of hard requirements, and that Figure 1 continues to illustrate the relation between the involved adaptor specifications.

Finally it is also worth noting that hard requirements can be used to specify strict adaptation requests. Namely if all correspondence rules of a submitted specification S are hard requirements, then adaptor generation is constrained to produce a boolean result: Either S itself can be returned as a contract, or no adaptation will be proposed.

8 Concluding remarks

We have analysed the notion of adaptor specification under different perspectives in order to contribute to set a theoretical foundation for the adaptation of heterogeneous components presenting mismatching interaction behaviour. We believe that the definition of a formal semantics for adaptor specifications contributes to provide a clearer understanding and to ease a proper usage of the software adaptation methodology. In particular, a precise semantics of adaptor specifications is obviously necessary to avoid possible ambiguities in the process of adaptation trading, as well as to clarify the meaning of soft adaptation and of hard requirements.

A number of practice-oriented studies have analysed different issues encountered in (manually) adapting a third-party component for using it in a (possibly radically) different context (e.g., see [5,6,10]). On the other hand, while component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering, quite a few efforts have been devoted to develop its foundational aspects.

A formal foundation for adaptation was set by Yellin and Strom in their seminal paper [9], which constituted the starting point for our work. They employed finite state machines for specifying component behaviour, and formally introduced the notion of *adaptor* as a software entity capable of enabling two

components with mismatching behaviour interoperate. They used finite state grammars to specify interaction protocols between components, to define a relation of compatibility, and to address the task of (semi-)automatic adaptor generation. Some significant limitations of their approach derive from the expressiveness of the notation used, such as the impossibility of representing internal choices or parallel composition of behaviour. Moreover, the asymmetric meaning they gave to input and output actions made it necessary the use of *ex-machina* arbitrators to control system evolutions. Last, but not least, adaptor specifications in [9] allowed only to express one-to-one relations between actions, a severe expressiveness bound when facing non-trivial protocol adaptations as discussed in [1].

A category theory approach to component adaptation was presented in [11]. Component connections were defined by defining morphisms between the components'actions. While the morphisms of [11] may resemble our specifications, they can express only syntactic adaptations (viz., name translations), and cannot be used to resolve mismatchings in the interaction protocols.

Finally, we foresee different lines for future investigations. A natural direction is to extend the formal treatment of adaptor specifications to consider data dependencies across different actions, which may be defined by introducing action parameters in correspondence rules. Another interesting extension is to consider multi-party adaptations, rather than pair-wise adaptations. Notice that the syntax of adaptor specifications can be naturally lifted to deal with n components, by simply interpreting the operators $\diamondsuit$ and $\Diamond\!\!\!\!\!\Diamond$ as polyadic rather than diadic, allowing rules of the form:

$$a \diamondsuit b, c \Diamond\!\!\!\!\!\Diamond d\ ;$$

to specify correspondences among three parties.

Another interesting direction is to further develop the usage of specifications for adaptor trading. For instance, the definition of suitable metrics would allow to quantitatively evaluate the distance between the requested and the proposed adaptation, including the degree of sub-servicing proposed in the case of soft adaptation, and to quantitatively compare different adaptations.

Acknowledgements

The work of A. Brogi has been partly supported by MIUR Project NAPOLI. 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.

References

[1] 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.

[2] A. Brogi, C. Canal, and E. Pimentel. Soft component adaptation. To appear in *Electronic Notes in Theoretical Computer Science (ENTCS)*, 85(3), 2003.

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

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

[5] S. Ducasse and T. Richner. Executable connectors: Towards reusable design elements. In *ESEC/FSE'97*, LNCS 1301. Springer, 1997.

[6] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

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

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

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

[10] K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. SEI Series in Soft. Engineering, 2001.

[11] M. Wermelinger and J.L. Fiadeiro. Connectors for mobile programs. *IEEE Transactions on Software Engineering*, 24(5):331–341, 1998.