On the semantics of software adaptation

Antonio Brogi a,∗, Carlos Canal b, Ernesto Pimentel b

a Department of Computer Science, University of Pisa, Italy
b Department of Computer Science, University of Málaga, Spain

Received 30 January 2004; received in revised form 24 May 2005; accepted 10 October 2005
Available online 4 April 2006

Abstract

The problem of adapting heterogeneous software components that present mismatching interaction behaviour is one of the crucial problems in Component-Based Software Engineering. A promising approach to solve this problem is based on an adaptation methodology relying on extending component interfaces with protocol information which describes their interaction behaviour, and using a high-level notation to express the intended connection between component interfaces. The adaptor specification defines a component-in-the-middle capable of making two components interact successfully, according to certain constraints. The aim of this paper is to contribute to setting a theoretical foundation for software adaptation following this approach. A formal analysis of adaptor specifications is presented, and their usage to feature different forms of flexible adaptations is illustrated.

Keywords: Software adaptation; Formal methods

1. Introduction

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering [8,15]. The possibility of adapting off-the-shelf software components to work properly within new applications is a must for the development of a true component marketplace, and for component deployment in general [7]. Available component-oriented platforms feature Interface Description Languages (IDLs) to address software interoperability at the signature level. IDLs are a sort of lingua franca for specifying the functionalities offered by heterogeneous components that were developed in different languages. While IDL interfaces allow one 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 [19].

In our previous work [4,5], 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:

(1) 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.
(2) **Adaptor specification.** Adaptor specifications are simply expressed by sets 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.

(3) **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 been proven to succeed in a number of diverse situations [4,5], where a suitable adaptor is generated to support the successful interoperation of heterogeneous components presenting mismatching interaction behaviour. 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. The separation of adaptor specification and adaptor derivation thus 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.

It is worth remarking that the formalism employed for expressing adaptor specifications is deliberately simple. Indeed, as pointed out in [4], the objective is to ease as much as possible the task of specifying the needed adaptation, which simply amounts to establishing signature correspondences. Behavioural information will be taken into account during the adaptor derivation phase, which carries the burden of devising an adaptor capable of letting the protocols of the two components interoperate successfully.

While adaptor specifications have been employed thoroughly in [4] 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 the semantics of adaptor specification. In particular, after presenting a simple motivating example to illustrate the adaptation methodology (Section 2), we will focus on adaptor specifications and start by presenting their precise syntax (Section 3). Then, we will analyse the formal semantics of adaptor specifications (Section 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. Next, we will move (Section 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 whole adaptation process. Then, a more flexible or soft form of adaptation will be formally presented (Section 6), where the notion of sub-servicing is employed to weaken the initial adaptor specification when the latter cannot be fully satisfied. Next, the possibility of expressing hard requirements in adaptor specifications will be illustrated (Section 7), and their effect on adaptor generation described. Finally, several significant related works will be discussed (Section 8), and some concluding remarks will be drawn (Section 9).

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 [4,5] for more significant examples of software adaptation.

### 2. An example of software adaptation

To provide the context, we first illustrate a simple example of software adaptation. Following [4], we assume that component interfaces include interaction patterns that describe the essential aspects of the finite behaviour that a (non-recursive) component may (repeatedly) show to the external environment. Syntactically, these patterns are terms of a CCS-like process algebra.

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

$$\text{query?(). result!(). 0}$$

Now consider a client $Q$ that issues a query and waits for an answer, but may also decide to stop waiting, aborting the request. Suppose that the behaviour of $Q$ is expressed by the interaction pattern:

$$\text{request!(). (reply?(). 0 + \tau.abort!(). 0)}$$
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 (i.e., without deadlocks). 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 a number of rules establishing correspondences between actions of the two components. The natural specification of the adaptor for the example at hand is:

\[
\begin{align*}
&\text{query} \diamond \text{request}; \\
&\text{result} \diamond \text{reply}; \\
&\text{result} \diamond \text{abort}
\end{align*}
\]

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

Given an adaptor specification, a fully automated procedure [4] returns (if possible) an adaptor component that satisfies the specification and that lets the two components interoperate successfully. For instance, the process may return the adaptor:

\[
\text{request}?().\text{query}!().\text{result}?().(\text{reply}!().0 + \text{abort}?().0)
\]

### 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 \(\alpha_i\) and \(\beta_j\) 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 — which we will call *"the component on the left"* — while actions on the right side of rules refer to the other component — the *"component on the right"*. While in [4] adaptor specifications may include data dependences, 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. For instance the rule:

\[
a \diamond c
\]

is used to specify that, whenever the adaptor will perform one action *a* for matching one action of the component on the left, eventually it will have to perform one corresponding action *c*, or vice versa. Similarly, the rule:

\[
a, b \diamond c
\]

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

The adaptation needed to let the two parties interoperate may have to cope with asymmetries, typically when an action in one of the components does not have a correspondence 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 the component on the left, there is no corresponding action to be performed with respect to the component on the right.

Notice that the rules above 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:

\[
a \diamond b
\]

is satisfied both by the adaptor \(a.b.a.b.0\) and by the adaptor \(a.a.b.b.0\).
Hence, the syntax of adaptor specifications features a second operator $\otimes$ to express tighter correspondences among (sets of) actions in a rule. In particular, the operator $\otimes$ does not allow the interleaving of different occurrences of actions from a correspondence rule. Consequently, the rule:

$$ a \otimes b $$

is now satisfied by the adaptor $a.b.a.b.0$ but not by the adaptor $a.a.b.0.

An adaptor specification is hence a (finite) set of rules, separated by ";". Notice that the syntax for rules allows non-determinism in the specification of action correspondences. For instance, a specification such as:

$$\begin{align*}
\{ a \otimes b ; \\
a \otimes c, d ; \\
a \otimes
\end{align*}$$

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

4. Semantics of adaptor specifications

This section is devoted to analysing the formal semantics of adaptor specifications. We first show (Section 4.1) the process calculus that we are going to consider as the basic formalism to define the semantics of adaptor specifications (Section 4.2) which will be characterized by a set of processes describing the interaction behaviour of the adaptor components capable of featuring the desired adaptation. We then show (Section 4.3) that this process-based 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. Finally, in Section 4.4, an alternative characterisation of the semantics of an adaptor specification is given in terms of (an abstraction of) the process traces that satisfy it.

4.1. A CCS-like process calculus

The process calculus that we are considering in this paper is a version very close to CCS. Indeed, it is a subset of CCS with explicit silent actions. A silent action, denoted by $\tau$, describes an internal action autonomously made by an agent. The syntax of this simple process calculus is given by:

$$P ::= 0 \mid a.P \mid \tau.P \mid P + P \mid P|P$$

where $a$ ranges over a set of atomic actions, $+$ denotes the non-deterministic choice, and $|$ represents the parallel composition.

Since processes are used here to describe intensional behaviour (of both adaptors and properties) as we will see below, synchronization is not allowed within processes. Formally, the following non-synchronizing semantics of processes is used:

$$\begin{align*}
a.P &\rightarrow^a P \\
\tau.P &\rightarrow^\tau P \\
P &\rightarrow^\tau P' \\
P + Q &\rightarrow^\tau P' \\
P|Q &\rightarrow^\tau P'|Q
\end{align*}$$

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

4.2. Process-based semantics

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:

$$S = \{ a \otimes b ; \otimes c \}$$
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 one action \(a\) (respectively, \(b\)), then it will have to perform eventually one action \(b\) (respectively, \(a\)) — i.e., actions \(a\) and \(b\) must be performed in pairs, though they may freely interleave. Notice that the number of pairs of \(as\) and \(bs\) is not determined, though process \(R_1\) may eventually stop via an internal \(\tau\) move. Similarly, property \(R_2\) simply states that the adaptor may perform the action \(c\) an arbitrary number \(n \geq 0\) of times.

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

\[
d \overset{\diamond}{=} e;
\]

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 that are \textit{simulated} by the process \(\Pi(S)\).

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

**Definition 1.** A process \(P\) is \textit{simulated} by \(Q\) (\(P \preceq Q\)) if and only if:

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

We will use the above notion of process simulation to characterise the set of processes that satisfy a given adaptor specification \(S\). It is worth observing that the notion of simulation considered is slightly different to standard simulation. In fact, the first rule exactly corresponds to strong simulation, but the second rule identifies the inaction process \((0)\) with any process exhibiting a finite number of \(\tau\) transitions. Thus, the \(\preceq\)-simulation is equivalent to the union of strong simulation and the equivalence class of process \((0)\) with respect to the standard weak simulation. We have adopted this notion of simulation because component interfaces are expressed by finite interaction patterns, and we are interested in \textit{finite} processes capable of adapting such patterns. (Indeed, the above notion of simulation characterises correctly 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 instantiate Definition 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 \(P \xrightarrow{a_1} \cdots \xrightarrow{a_n} P'\) and \(P' \equiv 0\), then \(Q \xrightarrow{a_1} \cdots \xrightarrow{a_n} \tau^* Q'\) 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 2.** A process \(P\) \textit{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: \([S]\) = \{\(P\) | \(P \preceq \Pi(S)\)\}.

Notice that, in general, \([S]\) denotes an infinite set of processes. For instance, consider again the specification:

\[
\{\text{a} \diamond \text{b}; \diamond \text{c}\}.
\]

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. On the other hand, $\|S\|$ will not include, for instance, processes $a.b.a.0, c.b.0, or d.0$.

The above denotation $\|\|$ directly induces an ordering on adaptor specifications.

**Definition 3.** Given two adaptor specifications $S_1$ and $S_2$, we write:

$$S_1 \leq S_2 \text{ if and only if } \|S_1\| \subseteq \|S_2\|.$$ 

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

$$S_1 = \{a \diamond \} \leq S_2 = \{a \diamond ; \diamond b\}$$

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 us to perform $b$.

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:

$$\begin{align*}
{l_1} & \diamond \ ; \\
{\vdots} & \\
{l_m} & \diamond \ ; \\
& \diamond r_1 \ ; \\
& \vdots \\
& \diamond r_n \\
\end{align*}$$

(where $V = \{l_1, \ldots, l_m, r_1, \ldots, r_n\}$ is the vocabulary of all actions considered) is the largest\(^1\) specification in the $\leq$-ordering.

### 4.3. Properties of adaptor specifications

It is now worth stating some properties of adaptor specifications, which help to understand their meaning and usage. The first property below (Proposition 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 $\otimes$ 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.** Let $S$, $S_1$, and $S_2$ be adaptor specifications, let $r$ be a correspondence rule, and let $\alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_n$ be actions. Then:

1. $S \leq S \cup \{r\}$;
2. $\{\alpha_1, \ldots, \alpha_m \otimes \beta_1, \ldots, \beta_n\} \leq \{\alpha_1, \ldots, \alpha_m \diamond \beta_1, \ldots, \beta_n\}$;
3. $S_1 \leq S_2$ implies that $S_1 \cup S \leq S_2 \cup S$.

**Proof.**

1. The property follows immediately by the definition of process simulation. Consider a process $P \in \|S\|$, that is, $P \preceq \Pi(S)$. It is easy to see that $P \preceq \Pi(S \cup \{r\})$, since any trace of $\Pi(S)$ is also a trace of $\Pi(S \cup \{r\})$. Hence, if $P$ evolves into $P'$ with a trace $t$, and $P' \equiv 0$, then also $\Pi(S \cup \{r\})$ can evolve with the same trace $t$ into a process $Q$ such that $Q \xrightarrow{t} Q'$ and $Q' \equiv \Pi(\{r\})$. Since $\Pi(\{r\}) \xrightarrow{t} 0$ by construction of $\Pi()$, then we have that $\|S\| \subseteq \|S \cup \{r\}\|$.

\(^1\) Note that there are infinite specifications that are equivalent to this largest specification. They can be obtained by adding arbitrarily any other rule (for instance, $l_i, l_i, \ldots, l_i \diamond$) to such a specification.
Corollary 5. Let $S_1$, $S_2$, and $S$ be adaptor specifications. Then:

1. $S_1 \leq S$ iff $S_1 \cup S \leq S$;
2. $(S_1 \leq S$ and $S_2 \leq S)$ implies that $S_1 \cup S_2 \leq S$;
3. $\text{lub}(S_1, S_2) = S_1 \cup S_2$.

Proof.

1. Immediate by properties (1) and (3) of Proposition 4.
2. By property (3) of Proposition 4, we have that $S_1 \cup S_2 \leq S \cup S_2$ (since $S_1 \leq S$). Since $S_2 \leq S$, we also have — by the previous property (1) of this same Corollary — that $S \cup S_2 \leq S$. Hence, $S_1 \cup S_2 \leq S$.
3. Clearly, $\text{lub}(S_1, S_2) \leq S_1 \cup S_2$ by property (1) of Proposition 4. Now let $S$ be an upper bound for both $S_1$ and $S_2$.

By the previous property (2) of this same Corollary, we have that $S_1 \cup S_2 \leq S$, hence $S_1 \cup S_2$ is the lowest upper bound for $S_1$ and $S_2$. □

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

Definition 6. Let $S_1$ and $S_2$ 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 $S_2$ when $S_2$ is an extension of $S_1$.

Obviously, by Proposition 4, if $S_1$ is a reduction of $S_2$, then $S_1 \leq S_2$. Notice, however, that, in general, the converse is not true. For instance,

$$\{a \otimes b\} \leq \{a \triangle b\}$$

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

Finally, Proposition 7 below illustrates the meaning of rules establishing many-to-many correspondences among actions. More precisely, the first property of Proposition 7 states that splitting a correspondence rule in two parts weakens a specification, that is, increases the number of adaptor specified — or that, dually, joining two correspondence rules strengthens a specification, that is, reduces the number of adaptors specified. (To see that the converse does not hold, consider, for instance, the specifications $S_1 = \{a, b \triangle c, d\}$ and $S_2 = \{a \triangle c; b \triangle d\}$, and observe that $a.c.0 \in [S_2]$ while $a.c.0 \not\in [S_1]$.) The second property of Proposition 7 instead illustrates how the meaning of a rule establishing a many-to-many correspondence among actions can be expressed in terms of the semantics of its elements. More precisely, a rule of the form $\overline{\alpha}, \overline{\beta} \triangle \overline{\gamma}, \overline{\delta}$ specifies the set of adaptors which perform the actions $\overline{\beta}$ whenever they perform the actions $\overline{\gamma}$ and $\overline{\delta}$, and which also perform the actions $\overline{\gamma}$ and $\overline{\delta}$.
Proposition 7. Let $\alpha, \beta, \gamma,$ and $\delta$ be (possibly empty) sets of actions. Let $\circ$ be either $\Diamond$ or $\Box$. Then:

(1) $[\alpha, \beta \circ \gamma, \delta] \subseteq [\alpha \circ \gamma; \beta \circ \delta]$;

(2) $[[\alpha, \beta \circ \gamma, \delta]] = [[\alpha \circ \gamma; \beta \circ \delta]] \cap [[\alpha \circ \gamma; \beta \circ \delta] \circ [[\alpha \circ \gamma; \beta \circ \delta]]$.

Proof.

(1) Immediate by definition of $\subseteq$, as any process simulated by $\Pi([\alpha, \beta \circ \gamma, \delta])$ is also simulated by $\Pi([\alpha \circ \gamma; \beta \circ \delta])$.

(2) ($\subseteq$) We observe that $[\alpha, \beta \circ \gamma, \delta] \subseteq [\alpha \circ \gamma; \beta \circ \delta]$ and $[\alpha, \beta \circ \gamma, \delta] \subseteq [\alpha \circ \gamma, \beta \circ \delta] \circ [[\alpha \circ \gamma; \beta \circ \delta]] \cap [[\alpha \circ \gamma, \beta \circ \delta] \circ [[\alpha \circ \gamma; \beta \circ \delta]]$.

($\supseteq$) For the case of $\circ = \Diamond$, we observe that, by definition of $[.]$, any process $P$ in $[[\alpha \circ \gamma; \beta \circ \delta]]$ performs the same number of $\beta$, $\gamma$, and $\delta$ actions, as well as an arbitrary (possibly different) number of $\alpha$ actions. However, if $P$ also belongs to $[[\alpha \circ \gamma, \beta \circ \delta] \circ [[\alpha \circ \gamma; \beta \circ \delta]]$ then, again by definition of $[.]$, it also performs the same number of $\alpha$, $\gamma$, and $\delta$ actions, and it hence belongs to $[[\alpha, \beta \circ \gamma, \delta]]$. The reasoning for the case of $\circ = \Box$ is analogous, as the combination of the two sets of constraints works the same in the case of non-interleaving correspondences. □

4.4. Weight-based characterisation of adaptor specifications

Definition 2 characterised the semantics of an adaptor specification $S$ as the set $[S]$ of processes that satisfy it. An alternative, lower-level characterisation of the semantics of an adaptor specification can be given in terms of the behavioural process traces that satisfy it. Such a lower-level characterisation can be exploited, for instance, to check here we consider specifications containing only the $\Diamond$ operator.²

Let us first introduce the notion of weight of a set of actions, a straightforward abstraction of the notion of trace, which considers only the number of times that each action is executed.

Definition 8. Let $A$ be a set of actions. A weight $\rho$ over the actions $A$ is a total mapping $\rho : A \rightarrow \mathbb{N}$ which associates each action in $A$ with a natural number.

The following definition formalises when a weight satisfies an adaptor specification. Intuitively speaking, a weight satisfies a specification $S$ if it is possible to distribute (via a function $\delta$) over the rules of $S$ the weight of each action so that every rule is satisfied.

Definition 9. Let $S$ be an adaptor specification, let $A_S$ be the actions in $S$, let $R_S$ denote the set of rules of $S$, and let $occ(a, r)$ denote the number of occurrences of an action $a$ in a rule $r$. A weight $\rho$ over the actions $A_S$ satisfies $S$ if and only if $\exists \delta : A_S \times R_S \rightarrow \mathbb{N}$ such that $\forall a \in A_S$:

(1) $\sum_{r \in R_S} \delta(a, r) = \rho(a)$;

(2) $\forall r \in R_S$:

(2.1) $\exists m \in \mathbb{N} : \delta(a, r) = m \times occ(a, r)$

(2.2) $\forall b \in A_S : \text{if } occ(a, r) > 0 \text{ and } occ(b, r) > 0, \text{ then } \frac{\delta(a, r)}{occ(a, r)} = \frac{\delta(b, r)}{occ(b, r)}$.

Condition (1) of Definition 9 simply ensures that the function $\delta$ effectively distributes over the rules of $S$ the whole weight of each action. Condition (2.1) ensures that $\delta$ assigns to a rule part of the weight of an action $a$ according to the number of occurrences of $a$ in $r$. Namely, if $a$ does not occur in $r$, then $\delta(a, r) = 0$. Otherwise, $\delta(a, r)$ must be a multiple of the number of occurrences of $a$ in $r$. For instance, consider the rule $r = a, a \Diamond b$. Then $\delta(a, r) = 6$ denotes that three pairs of actions $a$ are used to match three times rule $r$. Finally, condition (2.2) ensures that the constraints specified by the correspondence rules of $S$ are actually satisfied. For instance, considering again the rule $r$ above, if $\delta(a, r) = 6$ then $\delta(b, r)$ must be 3, since, according to the semantics of $r$, each couple of $as$ corresponds to a single $b$.

² The treatment of the $\Box$ operator would require a quite operational characterisation, and would make it similar to the operational definition of the adaptor generation algorithm presented in [4].
Consider, for instance, the specification:

\[
S = \{ a, b \diamond d; b, c \diamond e, e; \diamond f; \}
\]

The weight \( \rho = \{ a \mapsto 2, b \mapsto 3, c \mapsto 1, d \mapsto 2, e \mapsto 2, f \mapsto 3 \} \) satisfies \( S \), since there exists a distribution \( \delta \):

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 )</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( r_2 )</td>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( r_3 )</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

On the other hand, the weight \( \rho' = \{ a \mapsto 1, b \mapsto 1, c \mapsto 1, d \mapsto 2, e \mapsto 2, f \mapsto 3 \} \) does not satisfy \( S \), since there does not exist a suitable distribution \( \delta \) complying with Definition 9.

Finally, we prove that the weight-based characterisation of adaptor specifications is sound w.r.t. the process-based semantics given in Section 4.2, as stated by the following proposition. We denote by \( \rho_t \) the weight of a finite trace \( t \), which associates each action \( a \) with the number of occurrences of \( a \) in \( t \).

**Proposition 10.** Let \( S \) be an adaptor specification, and let \( P \in \llbracket S \rrbracket \). Then, for each trace \( t \) of \( P \), the weight \( \rho_t \) satisfies \( S \).

**Proof.** Recall that, by Definition 2, \( P \in \llbracket S \rrbracket \) iff \( P \) is simulated by \( II(S) \). Then, the proof is performed by construction, by incrementally building the distribution function \( \delta \) following the steps performed by the process \( II(S) \) to simulate \( P \). Namely, whenever \( II(S) \) performs a transition labelled \( a \) corresponding to the application of a rule \( r \), the value of \( \delta(a, r) \) is incremented. It is then easy to see that the \( \delta \) obtained satisfies, by construction, conditions (1) and (2.1) of Definition 9. Condition (2.2) is also satisfied by the \( \delta \) obtained, as the process \( II(S) \) would otherwise get stuck while trying to simulate \( P \), and this would contradict the initial hypothesis. \( \square \)

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 [4] derives (if possible) a concrete adaptor by exhaustively trying to build a component that satisfies the given specification while letting the components interoperate successfully. 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:

1. the client will successfully interoperate with the adaptor (viz., the client will not get stuck), and
2. 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; an example that the reader can find fully developed in [5]. Suppose that the client wishes to perform its `info` and `play` actions to request information on available movies and to view a movie, respectively, using its `data` action to receive data from the server. However, the commands for performing these operations are named in the server as `search`, `view`, `start`, and `stream`, respectively. Hence, the client submits the following adaptor specification $S$ (by establishing correspondences with the server actions), and supposes that it receives the following proposed contract $C$:

$$S = \{ info \land search ; play \land view, start ; data \land stream \}$$
$$C = \{ info \land search \}.$$

The straightforward reading of the proposed contract $C$ is that, while the server commits to letting 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.)

As in the example above, partial adaptation may simply consist of removing some correspondence rules from the specification submitted. Notice that, in such cases, the proposed contract $C$ is a reduction of the submitted request $S$, and thus $C \leq S$ by virtue of Proposition 4. Hence, the type of component adaptation that we have described so far, given a proposed adaptor specification $S$:

- either yields a (possibly partial) adaptation $C \leq S$,
- or 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 [4] has subsequently been extended in [5] to feature forms of soft adaptation. One of the key notions introduced in [5] 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 a service. For instance, in the video-on-demand service, offering a clip preview of a movie can be considered as a typical sub-service of offering the whole movie.

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

$$preview \sqsubseteq view$$

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. 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 component interfaces.

As one may expect, the introduction of sub-services notably increases the possibilities of successful adaptations, as an initial specification can be suitably weakened (when needed) by providing 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, the server may need to balance its current workload, or handle requests in terms of access rights as discussed in [5].

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 the presence of sub-service declarations can be described as follows:

1. 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.

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

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

**Definition 11.** Let $S$ be an adaptor specification, and let $\sqsubseteq$ 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 $\lozenge$ or $\otimes$) and where, for all $i$, $(\alpha'_i = \alpha_i$ or $\alpha'_i \sqsubset \alpha_i)$ and $(\beta'_i = \beta_i$ or $\beta'_i \sqsubseteq \beta_i)$.

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

\[
S = \{ \text{info} \lozenge \text{search} ; \text{play} \lozenge \text{view}, \text{start} ; \text{data} \lozenge \text{stream} \} .
\]

Suppose that the server interface contains the sub-service declarations:

- `preview` $\sqsubseteq$ `view`
- `advertise` $\sqsubseteq$ `search`
- `advertise` $\sqsubseteq$ `preview`

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

\[
S^* = \{ \text{info} \lozenge \text{search} ; \text{info} \lozenge \text{advertise} ; \text{play} \lozenge \text{view}, \text{start} ; \text{play} \lozenge \text{preview}, \text{start} ; \text{play} \lozenge \text{advertise}, \text{start} ; \text{data} \lozenge \text{stream} \}
\]

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’s policy, the server may return different contract proposals, such as:

\[
C_1 = \{ \text{info} \lozenge \text{search} ; \text{play} \lozenge \text{preview}, \text{start} ; \text{data} \lozenge \text{stream} \} \quad \text{or} \quad C_2 = \{ \text{info} \lozenge \text{search} ; \text{play} \lozenge \text{preview}, \text{start} ; \text{play} \lozenge \text{advertise}, \text{start} ; \text{data} \lozenge \text{stream} \}
\]
where $C_2$ indicates that some play requests will be adapted into previews while others will be adapted into advertisements. Notice that the server may actually return any partial adaptor for $S$, including, for instance:

$$C_3 = \{\begin{array}{l}
\text{play} \triangleleft \text{advertise}, \text{start} ; \\
\text{data} \triangleleft \text{stream}
\end{array}\}$$

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 12.** Let $S$ be an adaptor specification, and let $S^*$ be the sub-service expansion of $S$. Let $\sigma$ be a name substitution such that, if $T$ is an adaptor specification, then $T\sigma$ is obtained from $T$ by replacing some service name occurrences\(^3\) in $T$ with a corresponding super-service. Then, for each reduction $C$ of $S^*$, there exists a name substitution $\sigma$ such that $C\sigma$ is a reduction of $S$.

**Proof.** Since $S^*$ is a sub-service expansion of $S$ then, by Definition 11, $S^* = S \cup E$, where $E$ is the set of rules added to $S$ by the expansion. Any reduction $C$ of $S^*$ can consequently be decomposed as $C = C_S \cup C_E$, where $C_S$ is a reduction of $S$ and $C_E$ is a reduction of $E$. Define now $\sigma$ as the substitution that converts each rule of $C_E$ back into the rule of $S$ from which it was generated during the expansion that produced $S^*$. We have that $C_E\sigma \subseteq S$ and, since $C_S \subseteq S$, we also have that $C\sigma \subseteq S$. \square

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

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

$$C_1\sigma_1 = C_2\sigma_2 = \{\text{info} \triangleleft \text{search} ; \\
\text{play} \triangleleft \text{view}, \text{start} ; \\
\text{data} \triangleleft \text{stream}\} = S$$

while, for $\sigma_3 = \{\text{advertise} \leftrightarrow \text{view}\}$

$$C_3\sigma_3 = \{\text{play} \triangleleft \text{view}, \text{start} ; \\
\text{data} \triangleleft \text{stream}\} \leq S.$$  

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 satisfy strictly the initial adaptor specification. More precisely, the derived adaptor is described by a specification that is a reduction of the initial specification where some services have possibly been sub-serviced.

On the other hand, while adaptor derivation is free to 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.

Therefore, we extend the syntax of adaptor specifications to allow expressing hard requirements by introducing solid versions $\dagger$ and $\ddagger$ of the rule correspondence operators $\triangleleft$ and $\triangleright$. Intuitively speaking, a correspondence rule

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

\(^3\) Notice that, strictly speaking, name substitutions must be defined on name occurrences (rather than on names), as sub-servicing may be non-uniform in general. For instance, if $x' \subseteq x$ and $S = \{a \triangleleft x; b \triangleright x\}$, then $C = \{a \triangleleft x; b \triangleright x'\}$ is a reduction of $S^*$ where only the $b$ request for $x$ will be sub-serviced with $x'$. 

in a specification $S$ states that such a 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 use of hard requirements, consider again the video-on-demand example and suppose that the client submits the specification:

$$S = \left\{ \text{info ▷ search ;} \right. \left\{ \text{play ◄ view, start ;} \right. \left\{ \text{data ◄ stream} \right\} \right\}.$$  

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.

1. 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.

2. 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 explained in the previous section.

It is worth noting that Proposition 12 continues to hold in the presence of hard requirements, and that Fig. 1 continues to illustrate the relation between the adaptor specifications involved.

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. Related work

Recently, Software Adaptation has achieved the status of a definite working area in the field of Software Engineering. There is a significant number of research works addressing adaptation issues, and many forums are including it among their topics of interest, if not specifically devoted to adaptation [9].

Apart from the previous work in the field of the authors, part of which have already been mentioned, other relevant works range from: (i) synthesis papers, trying to characterise the field, and also presenting a survey of the current
approaches within Software Adaptation; (ii) practice-oriented studies, reporting practical experiences of success (or failure) in adapting existing third-party components in scenarios different from those they were developed for; and (iii) fundamental approaches, trying to establish formal grounds for the development of software adaptation techniques. One of the first papers explicitly addressing the problems of integrating software components is [14], where architectural mismatch (considered in a very general and broad sense) is presented as being caused by the different assumptions that system components make about their own environment. These assumptions are almost always implicit, and quite often they conflict, making them extremely difficult to analyze before building the system. The paper states the need of specific techniques for the analysis of mismatch, and architectural adaptation, and suggests some possible lines of research in this field.

A general discussion of the issues of component interconnection, mismatch and adaptation can also be found in [3]. Similarly, [15] describes the issues and challenges surrounding component adaptation and surveys various approaches in the literature. In this work, different adaptation techniques are compared and evaluated by adapting an existing component in a sample application.

In their book on component-based development [21], Wallnau et al. state that there is a growing gap between the theory and the practice of software design. The theory largely assumes that the design task is to develop specifications for software components; in reality, however, most component-based development relies on pre-existing components, which have pre-existing specifications, and must be adapted before reuse. With more and more software being developed from commercially available components, it is increasingly critical to recognize the challenges and constraints inherent in such specifications, and to use specific and proven techniques for building component-based systems in a real working environment.

Among the practice-oriented studies, we must mention a number of works that have analysed some of the issues encountered in (manually) adapting a third-party component for using it in a (possibly radically) different context. For instance, in [13] the decomposition of a software system into components and connectors at the design stage is proposed as a way of describing and reasoning about complex software architectures. First-class connectors written using an object-oriented language are proposed as a first step towards making software architecture more explicit at the implementation level. Their connectors are run-time reusable entities which control the interaction of components and can express a wide repertoire of interaction relationships.

A different approach is that of [8], which addresses specific issues in product-line development, where adaptable components are defined as members of a family of similar software components that supports tailoring as an intrinsic aspect of reuse. Differences among the instances of a family are conceived as a set of feature decisions and represented as parameters of adaptability.

Similarly, DeLine’s proposal of flexible packaging [12] circumvents adaption by deferring some decisions about component interaction until system integration time. Although the proposal is interesting for addressing certain kinds of adaptation, it assumes the use of a specific methodology and its related programming language right from the beginning (when the reusable-to-be components are implemented), and is not suitable for adapting and reusing existing components developed by third parties.

On the other hand, while component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering, only a few efforts have been devoted to developing its foundational aspects. Nevertheless, a number of closely related works have been undertaken in the field of Software Architecture, in particular by Allen and Garlan [1], where the notion of a connector, considered as a sort of adaptor between software components, has been promoted to a first-class status in the design and formalized for describing behavioral specifications and analysing protocol mismatch. Other formal approaches for detecting interaction mismatches at the architectural level are presented for instance in [10,11,17].

Garlan’s work on Software Architecture and the formal definition of connectors is carried out in [18], where connector wrappers or adaptors are characterized as protocol transformations, as a way of analysing how they would affect the behaviour of the connectors that they wrap, and also to study their compositionality. While this work goes a step further along the path to defining the semantics of adaptation, their wrappers are constructed ad hoc, and the problem of automatic adaptor generation is not considered in [18], nor is it considered in any of the works mentioned so far.

The formal foundation for automatic adaptation was set by Yellin and Strom in their seminal paper [20], which constituted the starting point for our work. They employed finite state machines for specifying component behaviour, and introduced formally the notion of adaptor as a software entity capable of enabling the interoperability of two components with mismatching behaviour. 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 that they gave to input and output actions made the use of ex-machina arbitrators necessary for controlling system evolutions. Last, but not least, adaptor specifications in [20] allowed the expression only of one-to-one relations between actions, which is a severe expressiveness bound when facing non-trivial protocol adaptations as discussed in [4].

As pointed out in [20], the first step needed to overcome behavioural mismatch is to let behaviour information be explicitly represented in component interfaces. In this sense, process algebras feature a very expressive description of interaction protocols, and enable sophisticated analyses of concurrent systems. For these reasons, their use for the specification of component interfaces and for the analysis of component compatibility has been widely advocated.

Here, it is worth mentioning the work of Inverardi and Tivoli [16]. This proposal goes beyond specifying and analyzing a set of properties, addressing how to enforce certain behavioural properties (namely deadlock-freedom) out of a set of already implemented behaviors. The software architecture imposed on the assembly allows for the detection and recovery of component integration anomalies. Starting from the specification in CCS of the system to be assembled and of its properties, they develop a framework that automatically derives the glue code for the set of components in order to obtain a property-satisfying system. While, in this sense, their framework for adaptation is similar to ours, their adaptors are only able to enforce behaviour by cutting off undesired branches in the derivation tree generated by composing component protocols, while name translations are made between the two protocols involved in a synchronous way, that is, without being able or memorizing and reordering messages and parameters as in our proposal.

On the other hand, a serious drawback of employing process algebras is the inherent complexity of verification procedures, which inhibits their usability in practice. Hence, other formal approaches have been investigated, including, for instance, a category theory approach to component adaptation, presented in [22]. Component connections were defined by defining morphisms between the components’ actions. However, while the morphisms of [22] may resemble our specifications, they can express only little beyond syntactic adaptations (viz., name translations), and cannot be used to resolve more general mismatches in the interaction protocols.

9. Concluding remarks

Although related to many of the works mentioned above, the goal of this paper is somewhat different: we have analysed the notion of adaptor specification under different perspectives in order to contribute to the setting of 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 providing a clearer understanding and to easing 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.

It is worth mentioning that our proposal constitutes a modular and flexible approach for specifying the required adaptation between two software components. Indeed, the way in which we address adaptor specification when issues such as access rights and sub-services are involved respects the separation of concerns advocated by Aspect-Oriented Software Development (AOSD), since the specification of these issues is orthogonal to the specification of the adaptor itself.

Finally, we foresee different lines for future investigations. A natural direction is to extend the formal treatment of adaptor specifications to consider data dependences 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 lifted naturally to deal with $n$ components, by simply interpreting the operators $\diamond$ and $\Diamond$ as polyadic rather than diadic, allowing rules of the form:

$$a \diamond b, c \Diamond d$$

to specify correspondences among three parties.

Furthermore, our notation for adaptor specification has been kept deliberately simple. The correspondences between groups of actions in the components being adapted are declared by directly referring to the names of these actions, which are used as constants in the specification, therefore providing a static binding between the interfaces of
the components being adapted. These flat specifications have proven useful for correctly describing the adaptation required in many situations considered. However, a higher-order notation can be defined (see, for instance, [2]), with enhanced expressiveness and allowing the description of conditional and dynamic binding between components. Finally, another interesting direction is to develop further the usage of specifications for adaptor trading. For instance, the definition of suitable metrics [6] allows one to evaluate quantitatively the distance between the requested and the proposed adaptation, including the degree of sub-servicing proposed in the case of soft adaptation, and to compare quantitatively different adaptations.

References