Adapting Components with Mismatching Behaviours

Andrea Bracciali[1], Antonio Brogi[1], and Carlos Canal[2]

[1] Dipartimento di Informatica, Università di Pisa, Italia
[2] Depto. Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain

Abstract. Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering. We present a formal methodology for adapting components with mismatching interaction behaviours. The three main ingredients of the methodology are: (1) The inclusion of behaviour specifications in component interfaces, (2) a simple, high-level notation for expressing adaptor specifications, and (3) a fully automated procedure to derive a concrete adaptor from a given specification.

1 Introduction

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering (CBSE) [5,17,15]. The possibility for application builders to easily adapt off-the-shelf software components to work properly within their application is a must for the creation of a true component marketplace and for component deployment in general [4].

Available component-oriented platforms (e.g., CORBA [27], COM [8], JavaBeans [29], VisualStudio .NET [21]) address software interoperability by using Interface Description Languages (IDLs). The provision of an IDL interface defining the signature of the methods offered (and possibly required) by a component is an important step towards software integration. IDL interfaces highlight signature mismatches between components in the perspective of adapting or wrapping them to overcome such differences.

However, even if all signature problems may be overcome, there is no guarantee that the components will suitably interoperate. Indeed, mismatches may also occur at the protocol level, because of the ordering of exchanged messages and of blocking conditions [30], that is, because of differences in component behaviours. While case-based testing can be performed to check the compatibility of the behaviour of components, more rigorous techniques are needed to lift component integration from hand-crafting to an engineering activity.

The availability of a formal description of the interaction behaviour of software components is necessary in order to rigorously verify properties of systems consisting of large numbers of components that dynamically interact one another [10]. For instance, an application builder would like to be able to determine beforehand whether the inclusion of a third-party component may introduce a deadlock possibility into her application.

J. Bishop (Ed.): CD 2002, LNCS 2370, pp. 185–199, 2002.

In this paper, we focus on the problem of adapting components that exhibit mismatching behaviour. As we already pointed out, available component platforms do not provide suitable means for describing the interaction behaviour of a component. Consequently, behaviour mismatches can be only manually adapted.

The problem of component adaptation has been the subject of intensive attention in the last few years. A number of practice-oriented studies have been devoted to analyse different issues to be faced when adapting a third-party component for a (possibly radically) different use (e.g., see [14,12,18]). A formal foundation for component adaptation was set by Yellin and Strom in their seminal paper [31]. They used finite state machines (FSM) for specifying component behaviour, and introduced the notion of *adaptor* as a software entity capable of letting two components with mismatching behaviours interoperate.

The objective of this paper is to present a formal methodology for adapting components with possibly mismatching interaction behaviours. The three main aspects of the methodology are the following:

1. *Component interfaces.* We extend traditional IDL with a description of component behaviours. A component interface therefore consists of two parts: A signature definition (describing the functionalities offered and required by the component), and a behaviour specification (describing the interaction protocol followed by the component). Syntactically, signatures are expressed in the style of traditional IDLs, while behaviours are expressed by using a subset of π-calculus [23] — a process algebra which has proved to be particularly well suited for the specification of dynamic and evolving systems.
2. *Adaptor specification.* We present a simple notation for expressing a specification of an adaptor intended to feature the interoperation of two components with mismatching behaviours. The adaptor specification is given by simply stating a set of correspondences between actions and parameters of the two components. The distinguishing aspect the notation is to allow a high-level, partial specification of the adaptor.
3. *Adaptor derivation.* A concrete adaptor component is then automatically generated, given its partial specification and the interfaces of two components. This fully automated process exhaustively tries to build an adaptor that will allow the components to interoperate while satisfying the given specification. The advantage of separating adaptor specification and derivation is to automate the error-prone, time-consuming task of generating a detailed implementation of a correct adaptor, while simplifying the task of the (human) software developer.

In the rest of this paper, extended component interfaces are introduced in Sect. 2, the notation for adaptor specifications is described in Sect. 3, while Sect. 4 illustrates the automatic generation of adaptors from specifications. The applicability of the whole methodology is exemplified in Sect. 5, where a realistic case of adaptation between two components employing different file transmission protocols is analysed. Finally, Sect.6 is devoted to discuss related work and to draw some concluding remarks.

2 Component Interfaces

Components interfaces will be described in terms of *roles*. Typically, a role is an abstract description of the interaction of a component with any other component it is related to. Hence, a component interface will be represented by a set of roles, each one devoted to a specific facet of the behaviour of the component.

The specification of a role is divided into two parts. The first one describes the component at the signature level, and it is similar to traditional IDL descriptions. Instead, the second part will describe the behavior related with the role signature using a notation derived from process algebras:

```
role roleName = {
    signature input and output actions
    behaviour interaction pattern }
```

The signature interface of a component role declares a set of input and output actions. These actions can be seen as the set of messages sent and received by the role (representing the methods that the component offers and invokes, the values or exceptions returned, etc.). Notice that typically IDLs represent only the services that the component *offers* to its environment (that is, the set of its output actions), while we explicitly represent also the services *required* by the component, as a set of input actions.

Both input and output actions may have parameters, representing the data interchanged in the communication. Parameters can be typed in order to allow for type-checking. For our purposes it is enough to distinguish between just two types: Link –representing channels through which messages can be sent and received– and Data representing any other data value.

With respect to the behaviour interface, it is described by means of what we call an *interaction pattern* [3]. Intuitively speaking, an interaction pattern describes the essential aspects of the finite interactive behaviour that a component may (repeatedly) show to the external environment.

The language we use for describing these patterns is a variant of the synchronous π-calculus. Since the calculus allows link names to be sent and received as values, it has proved to be a very expressive notation for describing applications with changing topologies, such as open systems. In particular, we use a sugared subset the polyadic π-calculus [22], a generalized version the basic π-calculus in which tuples, and not only single names, can be sent along links. The set of behaviour expressions formally defined as follows:

$$\texttt{E ::= 0} \;\mid\; \texttt{a.E} \;\mid\; \texttt{(x)E} \;\mid\; [x=y]\,\texttt{E} \;\mid\; \texttt{E} \,||\, \texttt{E} \;\mid\; \texttt{E + E}$$

$$\texttt{a ::= tau} \;\mid\; \texttt{x?(d)} \;\mid\; \texttt{x!(d)}$$

The special process 0 represents inaction, while internal actions are denoted by tau. Input and output actions are respectively represented by x?(d) and x!(d), where x is the link along which the actions are performed and d is a tuple of names (either links or data), sent or received along x. Restrictions, like (x)E, represent the creation of a new link name x in an expression E.

There is also a matching operator, used for specifying conditional behavior. Thus, the pattern [x=y] E behaves as E if x=y, otherwise as 0. Finally, also nondeterministic choice (+) and parallel (||) operators are defined. The summation `E + E'` may proceed either to `E` or to `E'`. On the other hand, communication will only be allowed between expressions belonging to different components. Hence, `E || E'` consists of expressions `E` and `E'` acting in parallel but not synchronizing.

Notice that interaction patterns do not contain recursion. The reason is that they are intended to specify finite fragments of interaction as an abstract way of representing component behaviour. In order to show the implications of this choice, consider for instance a component `Reader` sequentially reading a file. File items are received with an action `read?(x)` — the end-of-file condition being represented by a special value `EOF`. Suppose that the component may decide to break the transmission at any time by sending an action `break!()`. This behaviour would be expressed in full (recursive) π-calculus as follows:

```
Reader = read?(x). ( [x!=EOF] Reader + [x=EOF] 0 ) + tau. break!(). 0
```

indicating the fact that the component will repeatedly present a `read?` action until either an `EOF` is received or it decides (by performing a `tau` action) to break the transmission. However, the (non recursive) interaction pattern representing this particular component will simply read:

```
read?(x). 0 + tau. break!(). 0          // R1
```

in which some aspects of the behaviour —like recursion and the alternatives after the `read?` operation— have been abstracted by *projecting* them over time, and collapsing repeated actions into a single one.

Indeed, trying to describe all the aspects of the behaviour of a distributed system in one shot unavoidably leads to complex formulations of low practical usability. Instead, we focus on descriptions of the *finite* concurrent behaviours, making the verification of properties more tractable. In some sense, the choice of considering simple non-recursive interaction patterns resembles the use of types in conventional programming languages. While type checking cannot in general guarantee the correctness of a program, it does eliminate the vast majority of programming errors [3]. Similarly, pattern compatibility guarantees *local* correctness, that informally reads as "a component, together with its environment, is granted to succeed in its *current* step". For instance, the interaction between an HTML client and a web-server via an HTML FORM may be verified by checking the compatibility of the finite protocols they follow, expressed as patterns. This, obviously is not sufficient to guarantee the global success of the possibly non-terminating client component running together with its open environment.

In fact, the framework we propose aims at defining a suitable model for open systems, allowing for practical, possibly on-the-fly, verification. In this sense, the finiteness of the approach aims both at dealing with the incompleteness inherent in open systems, by limiting the analysis to what it is currently observable in the system, and at providing computational tools that are not subject to an *unbound* explosion of computational complexity.

A component may be represented by more than one role or pattern. Consider now that our reader component copies to disk the received file, using actions `fwrite!` and `fclose!`. Again, its behaviour in recursive π-calculus is:

```
Reader' = read?(x). ([x!=EOF] fwrite!(x). Reader' + [x=EOF] fclose!(). 0)
```

```
        + tau. break!(). fclose!(). 0
```

Now, instead of writing a single (but in fact, more complex) pattern for representing the component, we will partition its behaviour into two independent roles: one for describing how it reads the file (which is the pattern R1 previously defined), and the other describing its interaction with the file system, represented by the pattern:

```
tau. fwrite!(data). 0 + tau. fclose!(). 0    // R2
```

Thus, we allow for a modular representation and analysis of behaviour. Each role represents the reader from the point of view of the component to which the role is connected. Hence, while the decision of sending either a fwrite! or a fclose! action is motivated in the reader by the reception of data or end-of-file, the role R2 succeeds to express the point of view of the file system, for which the reader component seems to decide freely to send either action.

The special characteristics of mobility which are present in the π-calculus allow the creation and transmission of link names which can be later used for communication. This determines that the signature interface of a π-calculus interaction pattern is not fixed (like in other process algebras or in object-oriented environments), but instead they can be extended by link-passing.

As an example of these extensible interfaces, consider the pattern below, which specifies the behaviour of a component accepting queries in which a specific channel for returning the requested fact is indicated. The component is also able to raise an exception due to internal reasons (here represented by a tau action):

```
query?(return). return!(fact). 0 + tau. exception!(). 0
```

The pattern indicates that initially the component presents a signature interface consisting only of the actions query?(Link return) and exception!(). However, *after* performing the first of these actions, the interface is enlarged, and also return!(Data fact) must be considered part of it.

The notation we use for this purpose is the operator '>' (read as "before") which explicitly represents dependencies between link names in the interface. For instance, the interface of the component described above will be written as:

```
role QueryServer = {
    signature
        query?(Link return) > return!(Data fact);
        exception!();
    behaviour
        query?(return). return!(fact). 0 +
        tau. exception!(). 0 }
```

which indicates that the action return! will be performed through the link received in the previous query? action.

3 Adaptor Specification

Adaptation, in its generality, is a hard problem which involves a large amount of domain knowledge and may require complex reasoning. Hence, our approach aims at providing a methodology for specifying the required adaptation between two components in a general and abstract way. Moreover, the description of the necessary adaptation will be used to automatically construct a third component, that we call *adaptor*, which is in charge of mediating, when possible, the interaction of the two components so that they can successfully interoperate. In this section we will illustrate a simple and abstract language which permits us to describe the intended *mapping* among the functionalities of the two components to be adapted.

We first observe that adaptation does not simply amount to substituting link names. Consider for instance a component P that requests a file by means of an url, and a repository Q that first receives the url and then returns the corresponding file. Their behaviour interfaces are, respectively:

```
request!(url). reply?(page). 0       // P
query?(address). return!(file). 0    // Q
```

The connection between request! and query?, and between reply? and return! could be defined by the substitution:

σ = {t1/request, t1/query, t2/reply, t2/return}

that allows their interoperation. Notice that, after applying the substitution, the communication between Pσ and Qσ would be direct and unfiltered, since they will share link names. However this contrasts with encapsulation principles as, in general, one would like neither to modify the components nor to allow the sharing of names (methods or links) between different components. Moreover, it appears clear that this kind of adaptation can solve only renaming-based mismatchings of very similar behaviours. In general, one is interested in adapting more different situations where, for instance, reordering and remembering of messages may be necessary.

We represent an adaptor specification by a mapping that establishes a number of rules relating actions and data of two components. For instance, the mapping expressing the intended adaptation for the previous example is written as:

```
M = { request!(url) <> query?(url);
      reply?(file) <> return!(file); }
```

The intended meaning of the first rule of M is that every time P will perform a request! output action, Q must perform a corresponding query? input action. The use of parameters url and file in the mapping explicitly states the correspondence among data. Parameters have a global scope in the mapping, so that all the occurrences of the same name, even if in different rules, refer to the same parameter.

Intuitively speaking, a mapping provides a minimal specification of an adaptor that will play the role of a "component-in-the-middle" between two components P and Q. Such adaptor will be in charge of mediating the interaction between P and Q according to the mapping specification. It is important to observe that the adaptor specification defined by a mapping abstracts away from many details of the components behaviours. The burden of dealing with these details is put on the (automatic) adaptor construction process, that will be described in the next section. For instance, the behaviour interface of an adaptor satisfying the specification given by mapping M is:

```
request?(url). query!(url). return?(file). reply!(file). 0
```

Such adaptor will maintain the name spaces of P and Q separated and prevent the two from interacting each other without its mediation. Observe that the introduction of such an adaptor to connect P and Q has the effect of changing their communication from synchronous in asynchronous. Indeed, the task of the adaptor is precisely to *adapt* P and Q together, not to act as a transparent communication medium between them.

We conclude this section by sketching the syntax and usage of mappings for specifying different types of adaptation. (A full example of adaptation will be described in Sect.5.)

- *Multiple action correspondence.* While the previous example dealt with one-to-one correspondences between actions, adaptation may in general require relating groups of actions of different components. For instance, consider two components P and Q involved in an authentication procedure. Suppose that P authenticates itself by sending first its user name and then a password. Q instead is ready to accept both data in a single shot. Their behaviour interfaces and the mapping M specifying the required adaptation are:

```
user!(me). passwd!(pwd). 0     // P
login?(usr, word). 0           // Q

M = {user!(me), passwd!(pwd) <> login?(me,pwd);}
```

  The mapping associates both output actions performed by P to the single input action performed by Q, indicating also the reordering of parameters to be performed by the adaptor.
- *Actions without a correspondent.* Adaptation must also deal with situations in which an action of a component does not have a correspondent in the other component. For instance, consider a component P that authenticates itself (actions usr! and passwd!), asks for the list of files which are present in a repository (dir! and getdir?), and then deletes a file (delete!). The repository server Q does not require a login phase, but it rather expects a password to be sent together with the invocation of each service it provides (ls? for listing files, and rm? for deleting a file):

```
user!(me). passwd!(pwd). dir!(). getdir?(list). delete!(file). 0  // P
ls?(password). return!(files). rm?(name, password). 0            // Q
```

  From the viewpoint of Q, authentication concerns are spread over the whole interaction. Moreover, notice that the parameter me is not requested while pwd/passwd is used more times by Q.
  In order to explicitly represent this conceptual asymmetry among the two components, and hence to facilitate the task of devising and reasoning about the high-level specification of a mapping, we have introduced the keyword none. The actions of a component which do not have a clear correspondent

in the other component may be associated with none. Hence, the following mapping states that the login phase of P has not correspondence in Q and also that the parameter pwd must be recorded for subsequent uses.

```
M = { user!(me),passwd!(pwd) <> none;
      dir!() <> ls?(pwd);
      getdir?(files) <> return!(files);
      delete!(file) <> rm?(file, pwd); }
```

- *Nondeterministic associations between actions.* A difficult case for adaptation arises when the execution of a component action may correspond to different actions to be executed by the other component. Indeed, in general each component may perform local choices to decide what action to execute next. In such cases, adaptation should take care of dealing with many possible combinations of actions independently performed by the two components.
  In order to feature a high-level style of the specification of the desired adaptation, we allow nondeterminism in the adaptor specification. For instance, suppose a component receiving a file by means of a single action read? while its counterpart may decide to send an action data!(x) or an end-of-file eof!(). The mapping will be specified by means of two separate rules:

  ```
  read?(x) <> data?(x);
  read?(EOF) <> eof!();
  ```

  The adaptor derivation process will be then in charge of building an actual adaptor capable of dealing with all the possible specified situations. Once more, our goal is to allow the adaptor specification to abstract away from implementation details, and to leave the burden of dealing with these details to the (automatic) adaptor construction process. The use of nondeterministic associations will be illustrated further in the example of Sect.5.

4 Adaptor derivation

In this section we sketch how a concrete adaptor can be automatically generated, starting from two roles P and Q, and a mapping M. The adaptor derivation is implemented by (an extended version of) the algorithm we developed for checking the correctness of an open context of components [3]. The goal of the algorithm is to build a process A such that:

1. P|A|Q is successful (i.e. all traces lead to a successful state, where both P, Q and A have reduced to 0), and
2. A satisfies the given mapping M, that is, all the action correspondences and data dependencies specified by M are respected in any trace of P|A|Q.

The algorithm incrementally builds the adaptor A by trying to eliminate progressively all the possible deadlocks that may occur in the evolution of P|A|Q. Informally, while the derivation tree of P|A|Q contains a deadlock, the algorithm extends A with an action α that will trigger one of the deadlock states:

- Such action α is chosen so as to match a dual action $\overline{\alpha}$ on which P or Q are blocked. Notice that the adaptor is able to match only *some* of those actions. For instance, it cannot match an input action $\overline{\alpha}$ if it has not yet collected enough information to build a corresponding action α that satisfies the data dependencies specified in M.
- Since there may be more than one "triggerable" action $\overline{\alpha}$, at each step the algorithm non-deterministically chooses one of them to match, and spawns an instance of itself for each possible choice. If there is no triggerable action, then the algorithm (instance) fails.
- Each instance maintains a set $\mathcal{D}$ of data acquired by matching output actions, a set $\mathcal{F}$ of actions to be eventually matched so as to respect the correspondences stated by the rules of the mapping M, and a set $\mathcal{L}$ of link correspondences in order to guarantee the separation of name spaces between the two roles.
- Each algorithm instance terminates when the derivation tree of P|A|Q does not contains deadlocks. If the set $\mathcal{F}$ of actions to be matched is empty, then the algorithm instance successfully terminates and it returns the completed adaptor. It fails otherwise.

The overall algorithm fails if all its instances fail. Failure implies that the patterns P and Q can not be adapted according to the mapping M. If at least one of the instances returns an adaptor, the algorithm non-deterministically returns one of the adaptors found. This non-determinism is due to the non-determinism naturally present in concurrent systems. The definition of a representative of the class of the returned adaptors is currently under study: the general case presents some difficulties since the adaptors for the same patterns may be structurally much different. For the aims of this paper, it is important to remind that every returned adaptor makes P|A|Q successful.

For instance, considering again the third example of Sect.3 regarding the file repository server, the algorithm constructs the following adaptor A:

```
A = user?(me). passwd?(pwd). dir?(). ls!(pwd).
      return?(files). getdir!(files). delete?(file). rm!(file,pwd). 0
```

It is easy to verify that the composition P|A|Q is deadlock free, and that A satisfies the mapping, both in terms of action correspondence and data dependencies (e.g., A forwards pwd and file to Q only after receiving them from P.)

Moreover, in case of successful adaptor generation, the automatic construction phase also returns some information on the constructed adaptor. For instance, for the example above, the algorithm notifies that some data (viz., the value me) will be lost during the mediation, and that the adaptor will repeatedly send to Q the user password pwd though it is sent only once by P.

5 An Example of Adaptation

Consider a typical FTP transmission in which a file is sent by a server to a client. The example is simplified to show only the relevant details, while hopefully keeping its realistic flavour.

In order to make a modular specification of the problem, we will consider two different interactions between the client and the server, using two roles for describing their behavior. First, we will describe the behaviour of the client and the server regarding how to create and close an FTP session, and also how to request the typical put and get services for transmitting a file. Second, we will describe the details of file transmission using a separate pair of roles.

Hence, we will have two role-to-role connections, each one specified by a different mapping, from which the corresponding adaptors will be produced. Let us consider the first pair of roles. The roles IServer and IClient below describe the interface of the server and the client regarding the use of FTP commands.

```
role IServer = {
    interface open?(Link ctrl);
              user?(Data name, Data password, Link ctrl);
              put?(Data filename, Link ctrl);
              get?(Data filename, Link ctrl);
              close?(Link ctrl);

    behaviour open?(ctrl).
              user?(name,password,ctrl).
                  ( put?(filename,ctrl). close?(ctrl). 0
                  + get?(filename,ctrl). close?(ctrl). 0
                  + close?(ctrl) .0)  }
```

The role IServer above indicates how, for opening a session, a socket (represented by the link ctrl) must be provided. This socket will be used both for identifying the source of next commands (allowing thus multiple simultaneous sessions), and also for control communication between the client and the server. Once the connection is opened, the client must identify itself with a name and password. Then, put and get commands for file uploading and downloading can be issued to the server. Finally, the connection is ended with close.

```
role IClient = {
    interface login!(Data usr);
              pass!(Data pwd);
              getfile!(Data file);
              logout!();

    behaviour login!(usr).
              pass!(pwd).
              getfile!(file).
              logout!(). 0 }
```

On the other hand, suppose that the role IClient specifies that the client will connect with a login message by which it sends its identity, followed by a its password in a separate message (however no control socket is provided). Then, the client will ask for a certain file, and finally logs out.

It is worth observing that, in spite of the different behaviours of the two components, their adaptation can be simply specified by the mapping:

```
M = { login!(usr), pass!(pwd) <> open?(new ctrl), user?(usr,pwd,ctrl);
      getfile!(file) <> get?(file,ctrl);
      logout!() <> close?(ctrl); }
```

The first rule of M establishes the intended correspondence between two pairs of actions of the components. The mapping also exploits the use of action parameters to specify data dependencies among different actions of the components. In particular the `ctrl` parameter is employed in all the three rules to specify the needed adaptation due to the fact that the client does not specify the control socket in its protocol. The special keyword `new` in the first rule is used to specify the need for the adaptor to create a new name to match the server `open?` input action. As shown in Section 4, this mapping will produce the suitable adaptor:

```
A = login?(usr). pass?(pwd). (ctrl) open!(ctrl). user!(usr,pwd,ctrl).
    getfile?(file). get!(file,ctrl). logout?(). close!(crtl). 0
```

which allow both components to interact successfully. Let us consider now how a file is transmitted once a get command is issued by the client. Typically, the server will create a separate thread (daemon) for the transmission of the file. Accordingly, we will use another pair of roles, namely `IGetDaemon` and `IGettingFile`, for representing this facet of the interaction between client and server, respectively:

```
role IGettingFile = {
    interface read?(Data x);
              break!();

    behaviour read?(x). 0 + tau. break!(). 0 }

role IGetDaemon = {
    interface ctrl!(Link data, Link eof) >
                  data!(Data x),
                  eof!();

    behaviour (data,eof) ctrl!(data,eof).
                  ( tau. data!(x). 0 + tau. eof!(). 0 ) }
```

The differences between the two roles are the following:

- Server action `ctrl!` does not have a correspondent in the client, reflecting the fact that while the server creates specific links for each file transmission, the client uses fixed, predefined links for the same purpose. Hence, a suitable mapping rule for this situation is:

  ```
  none <> ctrl!(data,eof);
  ```

- The action for reading each piece of the file is called `read?` in the client, while the corresponding action in the server is `data!`. This mismatch can be easily solved with the mapping rule:

  ```
  read?(x) <> data!(x);
  ```

- The server may indicate at any moment the end of the file being transmitted by sending an `eof!()`, but the client does not have a corresponding message. This situation can be dealt with by using a special value in message `read?`, thus allowing the client protocol to end:

  ```
  read?(EOF) <> eof!();
  ```

- The client can autonomously decide (because of its local choice) to break the transmission at any moment by sending a `break!()` message. This case is more difficult to adapt, since the server and the adaptor might have already engaged themselves in a pair of complementary `data` actions. This would violate the one-to-one correspondence between actions `read?` and `data!` expressed by the second rule of the mapping (and no adaptor could be produced). We can solve this problem by stating also rules for mapping a client `break!()` to the actions `read!` and `eof!` of the server. Hence, the whole mapping would be:

```
M = { none <> ctrl!(data,eof);
      read?(x) <> data!(x);
      read?(EOF) <> eof!();
      break!() <> data!(y);
      break!() <> eof!(); }
```

Notice that the above mapping specifies action correspondences in a nondeterministic way. For instance, the last two rules state that the execution of the `break!` action may correspond to either a `data!` action or to a `eof!` action on the server side. Similarly, the second and fourth rule specify that the execution of a `data!` output operation by the server may match either a `read?` or a `break!` operation autonomously performed by the client.

It is important to observe that allowing nondeterministic correspondences in the mapping features a high-level style of the specification of the desired adaptation. While the mapping simply lists a number of possible action correspondences that may arise at run-time, the adaptor derivation process will be in charge of devising the actual adaptor able to suitably deal with all the possible specified situations. The adaptor produced from the above mapping is:

```
A = ctrl?(data,eof).  ( data?(x). ( read!(x). 0 + break?(). 0 )
                       + eof?(). ( read!(EOF). 0 + break?(). 0 )
                       + break?(). ( data?(x). 0 + eof?(). 0 ) )
```

6 Concluding Remarks

Several authors have proposed to extend current IDLs in order to deal with behavioural aspects of component interfaces. The use of FSMs to describe the behaviour of software components is proposed for instance in [9,20,26,31]. The main advantage of FSMs is that their simplicity supports a simple and efficient verification of protocol compatibility. On the other hand, such a simplicity is a severe expressiveness bound for modelling complex open distributed systems.

Process algebras feature more expressive descriptions of protocols, enable more sophisticated analysis of concurrent systems [1,24,25], and support system simulation and formal derivation of safety and liveness properties. In particular, the π-calculus, differently from FSMs and other algebras like CCS, can model some relevant features for component-based open systems, like dynamic creation of new processes, dynamic reorganization of network topology (mobility), and local and global choices. The usefulness of π-calculus has been illustrated for describing component models like COM [13] and CORBA [16], and architecture description languages like Darwin [19] and LEDA [6].

However, the main drawback of using process algebras for software specification is related to the inherent complexity of the analysis. In order to manage this complexity, the previous work of the authors has described the use of modular and partial specifications, by projecting behaviour both over space (roles) [7] and over time (finite interaction patterns) [3]. The calculus presented in this paper preserves most of the above mentioned features.

A general discussion of the issues of component interconnection, mismatch and adaptation is reported in [2,12,14], while formal approaches to detecting interaction mismatches are presented for instance in [1,7,11]. The problem of software adaptation was specifically addressed by the work of Yellin and Strom [31], which constitutes the starting point for our work. They use 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 are related with the expressiveness of the notation used. For instance, there is no possibility of representing local choices, parallel composition of behaviours, or the creation of new processes. Furthermore, the architecture of the systems being described is static, and they do not deal with issues such as reorganizing the communication topology of systems, a possibility which immediately becomes available when using the π-calculus. In addition, the asymmetric meaning they give to input and output actions makes it necessary the use of *ex machina* arbitrators for controlling system evolution.

Another closely related work is that of Reussner [28], who proposes the extension of interfaces with FSMs in order to check correct composition and also to adapt non-compatible components. Protocols are divided into two views: the services the component offers, and those it requires from its environment. One limitation of the work is that these two views must be orthogonal, i.e. each time a service is invoked in a component it results the same sequence of external invocations, while this usually depends on the internal state of the component. It should be also noticed that in Reussner's protocols there is no indication of action signs, and only method invocation is represented, while our approach involves a more general setting in which any dialogue or protocol between components can be specified. Finally, adaptation is considered in this work only as *restriction* of behaviour; if the environment does not offer all the resources required, the component is restricted to offer a subset of its services, but no other forms of adaptation (like name translation, or treatment of protocol mismatch) is considered.

The main aim of this paper is to contribute to the definition of a methodology for the automatic development of adaptors capable of solving behavioural mismatches between heterogeneous interacting components.

Our work falls in the research stream that advocates the application of formal methods, in particular of process algebras, to describe the interactive behaviour of software systems. As shown for instance in [3,7], the adoption of π-calculus to extend component interfaces paves the way for the automatic verification of properties of interacting systems, such as the compatibility of the protocols followed by the components of the system.

While the proposed methodology lays a foundation for the automatic development of adaptors, we foresee several interesting further developments. The first we intend to address is the formal verification of properties of the generated adaptor, such as security properties, as suggested in [20,31]. In practice, such a verification would allow an application to check that its security policy will not be spoiled by the inclusion of a new (adapted) component.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–49, 1997.
2. J. Bosch. Adapting object-oriented components. In *2nd. International Workshop on Component-Oriented Programming (WCOP'97)*, pages 13–22. Turku Centre for Computer Science, September 1997.
3. A. Bracciali, A. Brogi, and F. Turini. Coordinating interaction patterns. In *ACM Symposium on Applied Computing (SAC'2001)*. ACM Press, 2001.
4. A.W. Brown and H.C. Wallnau. The current state of CBSE. *IEEE Software*, 1998.
5. G. H. Campbell. Adaptable components. In *ICSE 1999*, pages 685 – 686. IEEE Press, 1999.
6. C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Software Architecture*, pages 107–126. Kluwer, 1999.
7. C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41:105–138, 2001.
8. D. Chappell. *Understanding ActiveX and OLE.* Microsoft Press, 1996.
9. I. Cho, J. McGregor, and L. Krause. A protocol-based approach to specifying interoperability between objects. In *Proceedings of TOOLS'26*, pages 84–96. IEEE Press, 1998.
10. E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of concurrency–Reflections and Perspectives*, Lecture Notes in Computer Science, 803. Springer, 1994.
11. D. Compare, P. Inverardi, and A. L. Wolf. Uncovering architectural mismatch in component behavior. *Science of Computer Programming*, 33(2):101–131, 1999.
12. S. Ducasse and T. Richner. Executable connectors: Towards reusable design elements. In *ACM Foundations of Software Engineering (ESEC/FSE'97)*, number 1301 in LNCS. Springer, 1997.
13. L.M.G. Feijs. Modelling Microsof COM using π-calculus. In *Formal Methods'99*, number 1709 in LNCS, pages 1343–1363. Springer, 1999.
14. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.
15. D. Garlan and B. Schmerl. Component-based software engineering in pervasive computing environments. In *4th ICSE Workshop on Component-Based Software Engineering*, 2001.
16. M. Gaspari and G. Zavattaro. A process algebraic specification of the new asynchronous CORBA messaging service. In *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 495–518. Springer, 1999.
17. George T. Heineman. An evaluation of component adaptation techniques. In *2nd ICSE Workshop on Component-Based Software Engineering*, 1999.

18. S. Hissam K. Wallnau and R. Seacord. *Building Systems from Commercial Components.* The SEI Series in Software Engineering, 2001.
19. J. Magee, S. Eisenbach, and J. Kramer. Modeling darwin in the π-calculus. In *Theory and Practice in Distributed Systems*, number 938 in LNCS, pages 133–152. Springer, 1995.
20. J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Software Architecture*, pages 35–49. Kluwer, 1999.
21. Microsoft Corporation. .NET Programming the Web. http://msdn.microsoft.com.
22. R. Milner. The polyadic π-calculus: a tutorial. Technical report, University of Edinburgh, Octubre 1991.
23. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.
24. A. P. Moore, J. E. Klinker, and D. M. Mihelcic. How to construct formal arguments that persuade certifiers. In *Industrial-Strength Formal Methods in Practice.* Springer, 1999.
25. E. Najm, A. Nimour, and JB. Stefani. Infinite types for distributed objects interfaces. In *Proceedings of the third IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'99.* Kluwer, 1999.
26. O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition.* Prentice Hall, 1995.
27. OMG. *The Common Object Request Broker: Architecture and Specification.* Object Management Group. http://www.omg.org.
28. R. H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *34th Hawaiin International Conference on System Sciences.* IEEE Press, 2001.
29. Sun Microsystems. JavaBeans API specification. http://java.sun.com.
30. A. Vallecillo, J. Hernández, and J. M. Troya. New issues in object interoperability. In *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, number 1964 in LNCS, pages 256–269. Springer, 2000.
31. D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.