ARTICLE IN PRESS

Available at www.ElsevierComputerScience.com
POWERED BY SCIENCE @ DIRECT®

The Journal of Systems and Software xxx (2004) xxx–xxx

The Journal of Systems and Software

www.elsevier.com/locate/jss

A formal approach to component adaptation

Andrea Bracciali [a], Antonio Brogi [a,*], Carlos Canal [b]

[a] Department of Computer Science, University of Pisa, Via Buonarroti 2, Pisa 56127, Italy
[b] Department of Languages and Computer Science, University of Málaga, Malaga 29071, Spain

Received 20 October 2002; received in revised form 1 February 2003; accepted 2 May 2003

Abstract

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering (CBSE). We present a formal methodology for adapting components with mismatching interaction behaviour. 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 concrete adaptors from given high-level specifications.

1. Introduction

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering (CBSE) (Campbell, 1999; Heineman, 1999), and it has been the subject of increasing attention in the last few years. The possibility for application builders to easily adapt off-the-shelf software components to properly work within their application is a must for the creation of a true component marketplace and for component deployment in general (Brown and Wallnau, 1998).

Available component-oriented platforms (e.g., CORBA, COM, JavaBeans, .NET) address software interoperability typically using Interface Description Languages (IDLs) to specify the functionality offered (and required) by possibly heterogeneous software components. IDL interfaces are important for software integration, since they highlight signature mismatches between components, in view of their adaptation or wrapping. However, solving all signature problems does not guarantee that the components will suitably interoperate. Indeed, mismatches may also occur at the protocol level, due to the ordering of the messages exchanged, and also to blocking conditions (Vallecillo et al., 2000), that is, because of behaviour mismatches of the involved components. Other than case-based testing of the compatibility of components, more rigorous techniques are needed to lift their integration from hand-crafting to an engineering activity.

For instance, system developers would like to determine beforehand whether the inclusion of a third-party component may introduce a deadlock into the application under development. In order to rigorously verify properties of systems consisting of large numbers of dynamically interacting components, a formal description of the interactive behaviour of components is needed (Clarke et al., 1994).

In this paper, we focus on the problem of *adapting* mismatching behaviours that components may exhibit. A formal foundation for adaptation was set by Yellin and Strom in their seminal paper (Yellin and Strom, 1997). There, they used finite state machines for specifying component behaviours, and introduced formally the notion of *adaptor* as a software entity capable of letting two components with mismatching behaviour interoperate.

The aim of this paper is to present a formal methodology for behavioural adaptation, whose main aspects are the following.

1. *Component interfaces.* IDL interfaces are extended with a description of the behaviour of the components. Hence, an interface consists of two parts: A signature definition (describing the functionalities offered and required by a component), and a behaviour specification

[*] Corresponding author.
E-mail address: brogi@di.unipi.it (A. Brogi).

0164-1212/$ - see front matter © 2003 Published by Elsevier Inc.
doi:10.1016/j.jss.2003.05.007

(describing the interaction protocol followed by a component). While signatures are expressed in the style of traditional IDLs, behaviour specifications are expressed by using a subset of π-calculus (Milner et al., 1992), a process algebra well-suited for the specification of dynamic and evolving systems.

2. *Adaptor specification.* We present a simple notation for expressing the specification of an adaptor intended to feature the interoperation of two components. The adaptor specification consists of a set of correspondences between actions and parameters of the two components. The distinguishing aspect of the notation is that it produces a high-level, partial specification of the adaptor. The meaning of the adaptor specification can be formalised into a set of properties (expressed in π-calculus), which constrains the automatic derivation of correct adaptors.

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 separation of adaptor specification and derivation allows for automating the error-prone, time-consuming task of generating a detailed implementation of a correct adaptor, thus simplifying the task of the (human) software developer.

Component interfaces and the notation for adaptor specifications are described in Section 2 and Section 3, respectively. Section 4 describes automated adaptor generation. An example in Section 5 illustrates the whole methodology. Related work and concluding remarks are discussed in Section 6.

2. Component interfaces

Component interfaces consist of a set of *roles* (Canal et al., 1999). Each role is an abstract description of a specific facet of the behaviour that the component plays in its interaction with any other component it will be related to. The specification of a role is divided into two parts: (1) a description of the component at the signature level (as usually done by IDLs), and (2) a description of the component interactive behaviour:

role *role Name* = {
  signature *input and output actions*
  behaviour *interaction pattern* }

The signature interface of a role declares a set of input and output actions, that is, 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. Differently from typical IDLs, not only the services that the component *offers* to its environment (i.e., its output actions), but also the services *required* by the component (i.e., its input actions) are explicitly indicated. Both input and output actions may have parameters, representing the data exchanged in the communication. Parameters can be typed, allowing for type-checking, but for the purpose of this paper only two different types are used: `Data` and `Link`. The latter identifies link names which can be sent and received by the component, and then used for interacting with its environment, while `Data` refers to generic data (anything but links).

The behaviour description of a role consists of what we call an *interaction pattern* (Bracciali et al., 2001). Intuitively speaking, an interaction pattern describes the essential aspects of the *finite* interactive behaviour that a component may (repeatedly) show to its environment. These patterns are described by means of a sugared subset of the polyadic π-calculus, in which tuples, and not only single names, can be communicated. The π-calculus, allowing link names to be sent and received as values, has proved to be a very expressive notation for describing the behaviour of software components in applications with changing interconnection topology. Interaction patterns are defined as follows:

```
E : : = 0 | a.E | (x)E | [x = y]E | E||E | E + E
a : : = tau | x?(d) | x!(d)
```

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 parameters (either links or data), sent or received along `x`. Non-observable actions (also called silent actions) are denoted by `tau`. Actions are composed in expressions (processes), where `0` represents inaction. Restriction, e.g., `(x)E`, represents the creation of a new link name `x` in an expression `E`. The matching operator `[x = y]E` is used for specifying conditional behaviour: `[x = y]E` behaves as `E` if `x = y`, otherwise as `0`. Finally, non-deterministic choice (+) and parallel (‖) operators are defined: `E + E′` may proceed either to `E` or to `E′`, while `E‖E′` consists of expressions `E` and `E′` acting in parallel but, differently from the standard π-calculus parallel operator (|), not synchronising (only expressions of different components may communicate).

Notice that interaction patterns do not contain recursion, since they are intended to specify finite fragments of the interaction as an abstract way of representing component behaviours. In order to show the implications of this choice, consider, for instance, a reader component `R` that sequentially reads a file. File items are received via an action `read?(x)`, the end-of-file being represented by a special value `EOF`. Moreover, the component may decide to break the transmission at any time via an action `break!()`. Such a behaviour would be expressed in full (recursive) π-calculus as:

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

i.e., the component repeatedly presents a read? action until either an EOF is received, or it decides (by performing a tau action) to break the transmission. The encoding of this behaviour as a (non-recursive) interaction pattern, I1, is:

```
I1 = read?(x).0 + tau.break!().0
```

where some aspects of the behaviour, like recursion and the alternative after the read? action, have been abstracted by *projecting* them over time, 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 *finite* concurrent behaviours, making the verification of properties more tractable. In some sense, the choice of considering simple non-recursive interaction patterns resembles the introduction of types in programming languages. Even if type checking cannot in general guarantee the correctness of a program, it does eliminate the vast majority of programming errors. Similarly, even if the compatibility of a set of interaction patterns does not guarantee the correctness of a concurrent system, it can eliminate many errors in system assembly (Bracciali et al., 2001).

A component may exhibit more than one role or pattern. Consider the behaviour of a more complex reader, RW, which writes to disk the received file, using actions fwrite! and fclose!:

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

This behaviour can be partitioned into two independent roles: One for reading files, I1, and the other one, I2, for interacting with the file system:

```
I2 = tau.fwrite!(x).0 + tau.fclose!().0
```

Each role represents the reader from the point of view of the component to which the role is connected, facilitating a modular representation and analysis of behaviour. Indeed, I2 expresses the point of view of the file system, for which the reader seems to freely decide which action to output.

3. Adaptor specification

Adaptation 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. In this section we will illustrate a simple, high-level language for describing the intended *mapping* among the functionalities of two components to be adapted. This description will be used for the automatic construction of an *adaptor* that mediates the interaction of the two components.

We first observe that adaptation does not simply amount to unifying link names. Consider for instance a component P1 that requests a file by providing an url, and a server Q1 that first receives the url and then returns the corresponding file. Their interfaces are, respectively:

```
role P1 = {
   signature request!(Data url);
             reply? (Data page);
   behaviour request!(url).reply?(page).0
}
role Q1 = {
   signature query?(Data handle);
             return! (Data file);
   behaviour
     query?(handle).return! (file).0 }
```

The connection between request! and query?, and between reply? and return! could be defined by means of a substitution σ:

```
σ = { u/request, u/query, v/reply,
    v/return }
```

which allows the interaction of both components through links u and v. However, after applying the substitution, the communication between Pσ and Qσ would be direct and unfiltered, since they would share link names. Unfortunately, this contrasts with encapsulation principles as, in general, one would like neither to modify the components, nor to allow them to communicate directly, by-passing the adaptor. Moreover, this kind of adaptation can solve only renaming-based mismatching of very similar behaviours. We are instead interested in adapting less trivial mismatches where, for instance, reordering and remembering of messages is required.

Hence, 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 consists of the following two rules:

```
M1 = { request!(url) <> query?(url);
     reply?(file) <> return!(file); }
```

where as a convention, all the actions in the left hand side refer to the first of the components being adapted (in this case P1), while those in the right refer to the second one (here, Q1). The intended meaning of the first rule of M1 is that whenever P1 performs a request! output action, Q1 will eventually perform a corresponding query? input action. Similarly, the second rule indicates that whenever Q1 performs a return! action, P1 will eventually perform a reply? action. The parameters url and file explicitly state the correspondence among data. Parameters have a global scope in the mapping, so that every occurrence of the same name, even if in different rules, refers to the same parameter.

Intuitively speaking, the mapping M1 provides the minimal specification of an adaptor that will play the role of a "component-in-the-middle" between P1 and Q1, mediating their interaction according to the given 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 left to the (automatic) process of adaptor construction, that will be described in Section 4. For instance, the behaviour of an adaptor A1 satisfying the specification given by the above mapping M1 is:

```
A1 = request?(url).query!(url).
     return?(file).reply!(file).0
```

This adaptor will maintain the name spaces of P1 and Q1 separated and prevent them from interacting without its mediation. Observe that the introduction of such an adaptor to connect P1 and Q1 has the effect of changing their communication from synchronous to asynchronous. Indeed, the task of the adaptor is precisely to *adapt* P1 and Q1 together, not to act as a transparent communication medium between them.

Mappings can be used to specify different important cases of adaptation, as shown in the examples below.

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 both components. For instance, consider two components P2 and Q2 involved in an authentication procedure. Suppose that P2 authenticates itself by sending first its user name and then a password. Instead, Q2 is ready to accept both data in a single shot:

```
role P2 = { signature usr!(Data me);
                      pass! (Data pwd);
            behaviour usr!(me).pass!(pwd).0 }
role Q2 = { signature login?(Data acc, pin);
            behaviour login?(acc, pin).0 }
```

The required adaptation is specified by the mapping:

```
M2 = {usr!(me),pass!(pwd) <>
                            login?(me,pwd);}
```

which associates both output actions of P2 to the single input action of Q2. The mapping also illustrates the use of parameters (viz., me and pwd) to specify which data the adaptor must store for later use.

Actions without a correspondent. Adaptation must also deal with situations in which some actions of a component do not have a correspondent in the other one. For instance, consider a component P3 that features a printing service, waiting for requests for printing a number of copies of a document by means of an action printn?(doc,n), and another component Q3, which issues print requests in two steps: One for setting the number of copies, and one for actually printing the document. Their interfaces are, respectively:

```
role P3 = { signature printn?(Data doc, n);
            behaviour (...) }
role Q3 = { signature setCopies!(Data n);
                      print!(Data doc);
            behaviour (...) }
```

A suitable mapping for connecting P3 and Q3 can be defined as follows:

```
M3 = { none <> setCopies!(n);
    printn?(doc,n) <> print!(doc); }
```

The first rule of M3 indicates that the action setCopies! in Q3 does not have a correspondent in P3. The keyword none is used to explicitly represent this asymmetry between components.

Notice that in this example the situation is different from that described for multiple action correspondence. Indeed, the mapping M3 does not indicate whether Q3 will set the number of copies for each printing request, or whether a single setCopies! action will be issued for printing a given number of copies of several documents. However, a correct adaptor would be developed in either situation, depending on the actual behaviours of the two components (deliberately omitted in the example), which will be used for generating the adaptor, as we shall see in Section 4. Notice also that one could enforce the number of copies to be set for each printing request by specifying the mapping:

```
M3′ = { printn?(doc,n) <> setCopies!(n),
                          print!(doc); }
```

Indeed, M3′ specifies a multiple action correspondence so that the adaptor will ensure that Q3 will perform both a setCopies! and a print! output action for each printing request accepted by P3 with a printn? input action.

Non-deterministic action correspondence. A difficult case for adaptation arises when the execution of a component action may correspond to different alternative actions to be executed by the other component. 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 non-determinism in the adaptor specification.

For instance, consider a component `P4` sending a file by means of repeated `data!` actions. Suppose also that the corresponding reader component `Q4` receives the file with `read?` input actions, while it may also decide to interrupt the transmission at any time by issuing a `break!` action. Their interfaces are represented by the roles:

```
role P4 = { signature data!(Data n);
            behaviour data!(n).0}
role Q4 = { signature read?(Data m);break!( );
            behaviour read?(m).0 + tau.break!( ). 0}
```

The required adaptation can be simply specified by the mapping:

```
M4 = { data!(x) <> read?(x);
       data!(x) <> break!( ); }
```

The adaptor derivation process will be then in charge of building an adaptor capable of dealing with all the possible specified situations. Once more, our goal is to allow the adaptor specification to abstract away from many implementation details, and to leave the burden of dealing with these details to the (automatic) adaptor construction process. The use of non-deterministic action correspondences will be further illustrated in Section 5.

Name passing. The special characteristics of mobility which are present in the π-calculus allow for the creation and transmission of link names which can be later used for communication. Hence, we can address situations in which the topology of the communication between components is not necessarily static, but may change over time. This determines that the signature interface of a π-calculus interaction pattern is not fixed a priori (like in other process algebras or in object-oriented environments), but instead it can be extended by link-passing.

For instance, consider a situation very similar to the interaction described by components `P1` and `Q1`. There, we used predetermined links (`reply/return`) for the return value of the request, but it is also possible to indicate a newly created return link for each query:

```
role P5 = {
   signature request!(Data url, Link reply) >
        reply?(Data page);
   behaviour (reply) request!(url,reply).
           reply?(page).0}
role Q5 = {
   signature query?(Data handle, Link ret) >
          ret!(Data file);
   behaviour query?(handle,ret).ret!
           (file).0 }
```

Here, the situation is slightly different from that of `P1` and `Q1`. Role `P5` indicates that initially the component presents an interface consisting only of the action `request!`. However, after performing this action, the interface is enlarged with a new link name `reply`, which must be also considered part of it. This fact is indicated in the signature interface by using the operator '>' (read as "before") which explicitly represents the causal dependency between the parameter sent in the action `request` with the link used later for receiving the reply. Symmetrically for `Q5`, the link name received as the parameter `ret` in the `query?` input action will be used later for sending the return value. The mapping for connecting both components will be:

```
M5 = {
   request!(url,reply) <> query?(url,reply);
   reply?(file) <> reply!(file); }
```

4. Adaptor derivation

In the previous section, we have presented a simple notation for expressing a high-level specification of the adaptation needed to let two mismatching components interoperate. Given such a specification (mapping) `M`, and the interaction patterns `P` and `Q` of two components, a concrete adaptor (if any) is generated by means of a fully automated procedure. Intuitively speaking, such an adaptor will be a component-in-the-middle `A` such that:

(1) The parallel composition `P|A|Q` will not deadlock, and
(2) `A` will satisfy all the action correspondences and data dependencies specified by `M`.

Space limitations do not allow us to present here the algorithm for adaptor derivation in full details. We shall however summarise the essence of the algorithm w.r.t. points (1) and (2) above.

4.1. Deadlock elimination

The algorithm for adaptor generation has been obtained as a specialisation of the algorithm we developed (Bracciali et al., 2001) for checking the "so-far correctness" of open contexts of components. Such algorithm, given two patterns `P` and `Q`, returns a *completion* process

`A` (if any) such that the parallel composition `P|A|Q` will not deadlock.

To achieve (1), the algorithm tries to incrementally build a completion `A` by progressively eliminating all the deadlocks that may occur in the evolutions of `P|A|Q`. Because of its inherent non-deterministic nature, the construction has been naturally implemented in Prolog.

The algorithm is basically a loop which keeps track of the completion `A` constructed so far, as well as of the `last` action added to `A`. While the parallel composition `P|A|Q` is not deadlock-free, the algorithm tries to expand `A` with an action that will trigger one of the deadlocked states. Two cases are distinguished depending on whether `P|A|Q` may deadlock or not after executing the `last` action included in the completion.

(a) If `P|A|Q` may deadlock after executing action `last`, then an action `a` capable of triggering one of those deadlocked states is non-deterministically chosen (if any), and used to expand the completion as one of the possible actions following `last`. The construction process continues, being now `a` the `last` action included in the completion.

If there is no suitable triggering action, or if `P|A|Q` may both deadlock and succeed after executing action `last`,[1] then the algorithm backtracks to the state preceding the insertion of `last` in `A`.

(b) `P|A|Q` may deadlock, but no deadlock may occur after executing action `last` of the completion. In this case, there is no point in trying to expand further the completion “after” `last`. The algorithm hence tries to continue by considering the action that precedes `last` as the new `last` action.

To grasp the idea of how the algorithm works, consider for instance the simple case of the pattern `P=a!().(tau.b!().0+tau.c!().0)` and let `Q=0` for simplicity. The completion is initially empty and the parallel composition `P|Q` is stuck. Case (a) applies, and action `a?()` can be chosen to trigger the context, hence yielding the partial completion `A=a?().0` and setting `last` to `a?()`. The new context `P|(a?().0)|Q` presents now two deadlocks, both of them occurring after executing action `last`. Case (a) applies again, but there are now two possible triggers, namely `b?()` and `c?()`. Suppose that the algorithm (non-deterministically) chooses `b?()`, hence expanding the completion into `A=a?().b?().0`, being `b?()` the new `last`. The new context `P|(a?().b?().0)|Q` may still deadlock, but no deadlock may occur after executing `last` (viz., `b?()`). Case (b) then applies and the algorithm checks whether deadlocks may occur after executing `a?()`. This is indeed the case, hence the algorithm selects the only possible trigger `c?()` and adds it as a further choice after `a?()` in the completion, which now becomes

```
A=a?().(b?().0+c?().0)
```

The algorithm then terminates (as there are no other deadlocks in `P|A|Q`) and returns the constructed completion `A`.

To understand why the algorithm backtracks in case (a) when there is both a success and a deadlock after `last`, it is enough to consider the pattern `P=(a!().b!().0+a!().0)`, and let again `Q=0` for simplicity. It is easy to see that the only possible trigger `a?()` introduces both a successful and a deadlocked trace. But now no action can be added after `a?()`, as it would spoil the success. The algorithm will then backtrack and fail as there is no process `A` such that `P|A|Q` will not deadlock.

4.2. Mapping satisfaction

In order to derive *adaptors*, rather than simply completions, the algorithm of (Bracciali et al., 2001) needed to be specialised so as to take into account the constraints specified by the mapping. Basically this amounts to suitably constrain the way in which triggering actions are chosen during the incremental construction of the adaptor.

Technically this is done by exploiting the properties defined by the mapping, which define a set of constraints on the possible actions that the adaptor may perform at each moment. Each property is expressed as a π-calculus process, where actions are represented from the point of view of the adaptor, and combined according to the data dependencies implicitly stated by the corresponding mapping rule.

For instance, consider again the mapping `M1`. Its first rule indicated a one-to-one correspondence between actions `request!` and `query?`. Formally, this property can be represented by the following process:

```
R1=request?(url).(query!(url).0||R1)
     + tau.0
```

stating that whenever the adaptor performs a `request?` input action, then it will have to eventually perform a corresponding `query!` output action. Moreover, according to the data dependencies induced by parameters in the mapping rule, the adaptor should not perform an output action `query!(url)` until the requested `url` is received by means of the corresponding input action `request?(url)`. Finally, the process may eventually end via an internal `tau` move. Notice how the property refers to the point of view of the adaptor, not of the components. Thus, the sign (input/

[1] Indeed in the latter case any attempt to extend `A` with a new action “after” `last` would spoil those successes—see example below.

output) of the actions in the property is complementary with respect to the mapping.

When trying to solve an existing deadlock in `P|A|Q`, the algorithm may extend the current adaptor by choosing an action which is capable of triggering the deadlock while respecting the given properties. Consider for instance the patterns:

```
P1 = request!(url).reply?(file).0
Q1 = query?(q).return!(r).0
```

along with mapping `M1`, which is represented by the properties:

```
R1 = request?(url).(query!(url).0||R1)
     + tau.0
R2 = return?(file).(reply!(file).0||R2)
     + tau.0
```

The adaptor is initially empty, and the algorithm tries to extend it with an action capable of triggering the deadlocked composition `P1|Q1`. While both a `request?` and a `query!` action would trigger `P1|Q1`, only the former can be selected since the latter does not respect the expected behaviour specified by the properties. The algorithm then considers the new context `P1|(request?(url).0)|Q1` along with the properties updated according to the selection made:

```
R1′ = query!(url).0||R1
R2 = return?(file).(reply!(file).0||R2)
     + tau.0
```

The new context is still deadlocked and it might be triggered by both a `reply!` and a `query!` action. As the former does not respect properties `R1′` and `R2`, only the latter can be added to the current adaptor. Hence the algorithm will consider the new context `P1|(request?(url).query!(url).0)|Q1` along with properties `R1` and `R2` again. The construction will then continue similarly until all deadlocks will have been eliminated, while satisfying the properties imposed by the mapping. The returned adaptor will be

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

5. An example of adaptation

We illustrate now an example of application of the whole methodology. The problem to be solved regards the adaptation needed to support a typical FTP transmission in which a file is sent by a server to a client, when the two parties employ different protocols. Simplified in some aspects, the example shows only the relevant details, while hopefully keeping its realistic flavour.

In order to make a modular specification of the problem, we will use two roles for each component. First, we will describe the interaction for creating and closing a FTP session, and also for requesting a file transmission. Second, we will describe the details of file transmission using a separate pair of roles.

Each role-to-role connection needs a different mapping, from which a corresponding adaptor will be produced. The first pair of roles, `IServer` and `IClient`, describe the interface of the server and the client regarding the use of FTP commands.

```
role IServer = {
  signature open?(Link ctl);
            user?(Data name,pwd, Link ctl);
            put?(Data fn, Link ctl);
            get?(Data fn, Link ctl);
            close?(Link ctl);
  behaviour open?(ctl).user?(name,pwd, ctl).
            (put?(fn,ctl).close?(ctl).0
            + get?(fn,ctl).close?(ctl).0
            + close?(ctl).0))}
```

Role `IServer` indicates how, for opening a session, a socket (here named `ctl`) must be provided. This socket will be used both for identifying the source FTP commands (allowing thus multiple parallel sessions), and also for data transmission, as it will be shown in the second part of the example. Once the connection is opened, clients must authenticate themselves with a name and password. Then, `put` and `get` commands for file transmission can be issued. Finally, the connection can be ended with `close`.

```
role IClient = {
  interface login!(Data usr);
            pass!(Data pin);
            getfile!(Data file);
            logout!( );
  behaviour login!(usr).pass!(pin).
            getfile!(file).logout!( ).0}
```

On the other hand, the role `IClient` specifies that the client connects with a `login` message, followed by a password in a separate message (however no control socket is provided). Then, the client will ask for a certain file, and finally log out.

Despite the different behaviours of the two components, their adaptation can be simply specified by the following mapping:

```
MA = { login!(usr), pass!(pin)
       <> open?(ctl)user?(usr,pin,ctl);
       getfile!(file) <> get?(file,ctl);
       logout!( ) <> close?(ctl); }
```

The first rule of `MA` establishes the intended correspondence between log-in actions in both components, while the second rule adapts the file transmission commands. The third rule describes the correspondence between the log-out actions. The mapping also uses action parameters to specify data dependencies among different actions.

Starting from an action `login?(usr)`, the exploration of the derivation tree for constructing the adaptor is mainly guided by the behaviour described in `IClient`. As shown in Section 4, once an action on which `IClient` is deadlocked is matched by the adaptor, the mapping will trigger the matching of the corresponding action(s) in `IServer`, yielding in the end the adaptor:

```
AA = login?(usr).pass?(pin).
(ctl)open!(ctl).user!(usr,pin,ctl).
getfile?(file).get!(file,ctl).
logout?().close!(crtl).0
```

Notice that, even if the actual action ordering of the components'actions is not specified in the mapping, the exploration of the derivation tree of the two components implemented returns an adaptor which will enable both components to interoperate successfully, while at the same time respecting the mapping `MA`.

Let us now consider the file transmission phase. Typically, the server will create a separate thread (daemon) for the transmission of the file. In order to model this facet of the interaction, another pair of roles is used, `IGetDaemon` and `IGettingFile`.

```
role IGettingFile = {
   interface read?(Data x);
             break!();
   behaviour read?(x).0 + tau.break!().0 }
role IGetDaemon = {
   interface ctl!(Link data, Link eof) >
             data!(Data x), eof!();
   behaviour (data,eof)ctl!(data,eof).
              (tau.data!(x).0 + tau.eof!().0)}
```

The mapping for adapting both roles will be:

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

Its first rules establishes that server action `ctl!` does not have a correspondent in the client, reflecting the fact that while the server creates specific control links for each file transmission, the client uses fixed, predefined links for the same purpose.

Then, the second rule indicates that the reading of (a fragment of) a file is called `read?` in the client, while the corresponding action in the server is `data!`. However, the server may indicate at any moment the end of the file by sending an `eof!()`, while the client does not have a corresponding action. This mismatch is solved in the third rule by letting the adaptor forge a special value, `EOF`, and send it to the client, allowing the client role to terminate successfully.

In addition, the client can decide to break the transmission at any moment by sending a `break!()` message. This situation is slightly more difficult to adapt, since the server could not be able to react to such a message, being already engaged in transmitting a fragment of the file (`data!(x)` action), or in signalling the termination of the transmission (`eof!()`). Moreover, in this case the one-to-one correspondence between actions `read?(x)` and `data!(x)` expressed by the second rule of the mapping would be violated. However, the mismatch can be adapted by mapping client's `break!()` to both `read!(x)` and `eof!()` of the server as indicated by the last two rules of the mapping.

Notice that the mapping above specifies action correspondences in a non-deterministic way. Its 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!` operation by the server may match either a `read?` or a `break!` operation performed by the client.

It is important to observe again that allowing non-deterministic 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 is in charge of devising the actual adaptor able to suitably deal with all the possible specified situations.

Let us detail some of the steps of the construction of the adaptor for these two roles (to simplify the reading, we shall not list explicitly the properties derived from the mapping). Initially, the only possible trigger is the action `ctl?(data,eof)`. Once this action is chosen, we have four actions in which the roles are deadlocked: `data!`, `eof!`, `read?`, and `break!`. Suppose that the first selected for matching is `eof!`—the file is empty—, yielding the adaptor:

```
AB = ctl?(data,eof).eof?().0
```

At this point, following the mapping, the adaptor is expanded with the action `read!(EOF)`, so as to forward the `EOF` message to the client.

```
AB = ctl?(data,eof).eof?().
     read!(EOF).0
```

No more deadlocks can occur after executing `read!(EOF)` but the adaptor construction is not complete yet. For instance the client may autonomously decide to send a `break!()` before receiving any data from the server. The construction therefore continues by extending the adaptor with a branch capable to treat such a situation:

```
AB = ctl?(data, eof).eof?( ).
     (read!(EOF).0 + break?( ).0)
```

Again, no deadlocks can occur after executing `break?( )` but the process continues in order to complete the construction of the adaptor by building all the other needed alternatives, and finally returning the adaptor:

```
ctl?(data, eof).
(eof?( ).(read!(EOF).0 + break?( ).0)
+ data?(x).(read!(x).0 + break?( ).0)
+ break?( ).(data?(x).0 + eof?( ).0)
```

which adapts the roles `IGettingFile` and `IGetDaemon` respecting the mapping `MB`.

6. Concluding remarks

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 well-settled research stream which advocates the application of formal methods to describe the interactive behaviour of software systems. More specifically, we carry on the approach of enriching component interfaces with behavioural description for facilitating system analysis and verification in general (Inverardi and Tivoli, 2001; Magee et al., 1999; Najm et al., 1999) and behavioural mismatching detection in particular (Allen and Garlan, 1997; Canal et al., 2001; Compare et al., 1999), to cite but a few of the more closely related works. A distinguish feature of our approach consists of the adoption and use of a process algebra, namely a dialect of π-calculus, which allows for the automatic verification of a rich set of properties of interacting systems, mainly for what concerns the compatibility of component protocols.

Several proposals for extending IDLs with behavioural aspects are based on finite state machines, like, for instance (Yellin and Strom, 1997; Magee et al., 1999; Cho et al., 1998). The main advantage of finite state machines is that their simplicity supports a simple and efficient verification of protocol compatibility. However, 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 (Inverardi and Tivoli, 2001; Najm et al., 1999; Allen and Garlan, 1997; Moore et al., 1999), and support system simulation and formal derivation of safety and liveness properties, as also illustrated by the use of π-calculus for describing component models like COM (Feijs, 1999) and CORBA (Gaspari and Zavattaro, 1999), and architecture description languages like Darwin (Magee et al., 1995) and LEDA (Canal et al., 1999).

However, the main drawback of using full-fledged process algebras for software specification is related to the inherent complexity of their 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) (Canal et al., 2001) and over time (finite interaction patterns) (Bracciali et al., 2001), so as to ease automatic property verification. In this work we use a combination of both approaches.

A number of practice-oriented studies have analysed different issues encountered in (manually) adapting a third-party component for using it in a (possibly radically) different context (e.g., see Ducasse and Richner, 1997; Garlan et al., 1995; Wallnau et al., 2001). Besides, the problem of software adaptation was specifically addressed by the work of Yellin and Strom (1997), 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, i.e., the impossibility of representing internal choices, parallel composition of behaviours, creation of new processes, and the dynamic re-organisation of the communication topology of systems, a possibility which immediately becomes available when using the π-calculus. Also, the asymmetric meaning they give to input and output actions makes it necessary their use of *ex-machina* arbitrators for controlling system evolution. Finally, their mappings establish only one-to-one relations between actions, while our proposal address the issues of correspondence between actions, parameter storage and rearrangement in a more general setting.

A different approach is that of Wermelinger and Fiadeiro (1998), where software composition is addressed in the context of category theory. The connection between components is done by *superposition*, defining a morphism between actions in both components. Morphisms are similar to our mappings, though the kind of adaptation provided is more restrictive: They cannot remember previous actions or data, nor adapt different behaviours at the protocol level, limiting adaptation to a kind of name translation similar to that provided by IDL signature descriptions.

As it results from the comparison with significant related works appeared in the literature, and from the representative set of examples shown in this paper, our approach improves the capabilities of adapting components by combining expressiveness and effectiveness in a formally grounded methodology. Several promising lines of future research suggest to extend the framework for addressing issues like: Multiple-role adaptation, recovery strategies for adaptor construction failures, such as relaxing mapping constraints or devising partial adaptors, and the integration of the methodology in CBSE development tools.

References

Allen, R., Garlan, D., 1997. A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6 (3), 213–249.

Bracciali, A., Brogi, A., Turini, F., 2001. Coordinating interaction patterns. In: ACM Symposium on Applied Computing (SAC'2001). ACM Press.

Brown, A.W., Wallnau, K.C., 1998. The current state of CBSE. IEEE Software 5 (5), 37–47.

Campbell, G.H., 1999. Adaptable components. In: ICSE 1999. IEEE Press, pp. 685–686.

Canal, C., Pimentel, E., Troya, J.M., 1999. Specification and refinement of dynamic software architectures. In: Software Architecture. Kluwer, pp. 107–126.

Canal, C., Pimentel, E., Troya, J.M., 2001. Compatibility and inheritance in software architectures. Science of Computer Programming 41, 105–138.

Cho, I., McGregor, J., Krause, L., 1998. A protocol-based approach to specifying interoperability between objects. In: TOOLS'26. IEEE Press, pp. 84–96.

Clarke, E., Grumberg, O., Long, D., 1994. Verification tools for finite-state concurrent systems. In: A Decade of Concurrency, LNCS 803. Springer.

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

Ducasse, S., Richner, T., 1997. Executable connectors: towards reusable design elements. In: ESEC/FSE'97, LNCS 1301. Springer.

Feijs, L.M.G, 1999. Modelling Microsof COM using π-calculus. In: Formal Methods'99, LNCS 1709. Springer, pp. 1343–1363.

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

Gaspari, M., Zavattaro, G., 1999. A process algebraic specification of the new asynchronous CORBA messaging service. In: ECOOP'99, LNCS 1628. Springer, pp. 495–518.

Heineman, G.T., 1999. An evaluation of component adaptation techniques. In: ICSE'99 Workshop on CBSE.

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

Magee, J., Eisenbach, S., Kramer, J., 1995. Modeling Darwin in the π-calculus. In: Theory and Practice in Distributed Systems, LNCS 938. Springer, pp. 133–152.

Magee, J., Kramer, J., Giannakopoulou, D., 1999. Behaviour analysis of software architectures. In: Software Architecture. Kluwer, pp. 35–49.

Milner, R., Parrow, J., Walker, D., 1992. A calculus of mobile processes. Journal of Information and Computation 100, 1–77.

Moore, A.P., Klinker, J.E., Mihelcic, D.M., 1999. How to construct formal arguments that persuade certifiers. In: Industrial-Strength Formal Methods in Practice. Springer.

Najm, E., Nimour, A., Stefani, J.B., 1999. Infinite types for distributed objects interfaces. In: FMOODS'99. Kluwer.

Vallecillo, A., Hernández, J., Troya, J.M., 2000. New issues in object interoperability. In: Object-Oriented Technology, LNCS 1964. Springer, pp. 256–269.

Wallnau, K., Hissam, S., Seacord, R., 2001. Building systems from commercial components. SEI Series in Soft. Engineering.

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

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

Andrea Bracciali is currently Research Associate at the Department of Computer Science at the University of Pisa, Italy, where he is completing a Ph.D. degree in Computer Science. His research focuses on models for the analysis and verification of software systems in open environments.

Antonio Brogi is currently Associate Professor at the Department of Computer Science at the University of Pisa, Italy, where he received a Ph.D. in Computer Science in 1993. In the period 1994–95 he was Visiting Professor at the University of California, Los Angeles (U.C.L.A.), USA. His research interests include programming language design, coordination and adaptation of software components, and computational logic.

Carlos Canal is Associate Professor of Software Engineering at the University of Malaga (Spain), where he received his M.Sc. and Ph.D. degrees in Computer Science in 1993 and 2001, respectively. His research interests are Software Architecture and Component-Based Software Development, and in particular the application of formal methods to architectural specification, safe composition, and component adaptation issues.