A formal approach to component adaptation

Andrea Bracci ∗a, Antonio Brogi ∗a, and Carlos Canal b

aDepartment of Computer Science, University of Pisa, Italy
bDepartment of Languages and Computer Science, University of Málaga, Spain

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 CBSE [1,2], and it has been the subject of increasing attention in the last few years. The possibility for application builders to adapt easily off-the-shelf software components to work properly within their application is essential for the development of a true component marketplace and for component deployment in general [3].

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 order in which messages are exchanged, and also to blocking conditions [4], that is, because of behaviour mismatches of the components involved. Rather 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 [5].

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 [6]. There, they used finite state machines for specifying component behaviours, and introduced formally the notion of adaptor as a software entity capable of enabling two components with mismatching behaviour to 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 (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 [7], 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 permits the automation of 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 Sect. 2 and Sect. 3, respectively. Sect. 4 describes automated adaptor generation. An example in Sect. 5 illustrates the whole methodology. Related work and concluding remarks are discussed in Sect. 6.

### 2. Component interfaces

Component interfaces consist of a set of roles [8]. Each role is an abstract description of a specific facet of the behaviour of the component in its interaction with any other component to which 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 means of IDLs), and (2) a description of the component interactive behaviour:

```plaintext
role roleName = {
    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. Unlike with 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 [9]. 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:

```plaintext
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, nondeterministic choice (`+`) and parallel (`||`) opera-
tors are defined: $E + E'$ may proceed either to $E$ or to $E'$, while $E \parallel E'$ consists of expressions $E$ and $E'$ acting in parallel but, differently from the standard $\pi$-calculus parallel operator ($\parallel$), 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 $\text{read}(x)$, 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 $\text{break!}()$. Such a behaviour would be expressed in full (recursive) $\pi$-calculus as:

$$R = \text{read?}(x).([x=\text{EOF}] R + [x=\text{EOF}] 0) + \text{tao. break!}() \cdot 0$$

i.e., the component repeatedly presents a $\text{read?}$ action until either an EOF is received, or it decides (by performing a $\text{tao}$ action) to break the transmission. The encoding of this behaviour as a (non-recursive) interaction pattern, $I_1$, is:

$$I_1 = \text{read?}(x).0 + \text{tao. break!}() \cdot 0$$

where some aspects of the behaviour, like recursion and the alternative after the $\text{read?}$ action, have been abstracted by projecting them over time, collapsing repeated actions into a single one.

Indeed, trying to describe in full the behaviour of a distributed system unavoidably leads to complex formulations of low practical usability. Instead, we focus on descriptions of finite concurrent behaviours, making the verification of properties more feasible. 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 [9].

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 $\text{fwrite!}$ and $\text{fclose!}$:

$$RW = \text{read?}(x).([x=\text{EOF}] \text{fwrite!}(x).RW + [x=\text{EOF}] \text{fclose!}() \cdot 0) + \text{tao. break!}() \cdot \text{fclose!}() \cdot 0$$

This behaviour can be partitioned into two independent roles: One for reading files, $I_1$, and the other one, $I_2$, for interacting with the file system:

$$I_2 = \text{tao. fwrite!}(x).0 + \text{tao. fclose!}() \cdot 0$$

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, $I_2$ 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 difficult 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 introduce 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 $P_1$ that requests a file by providing an URL, and a server $Q_1$ that first receives the URL and then returns the corresponding file. Their interfaces are, respectively:

role $P_1 =$
- signature request!(Data url);
- reply?(Data page);

behaviour request!(url). reply?(page). 0 

role $Q_1 =$
- 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 P1 and Q1 would be direct and unfiltered, since they would share link names. Unfortunately, this conflicts 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 renumbering of messages is required.

Hence, we specify an adaptor by means of 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 states that whenever Q1 performs a return! action, P1 will eventually perform a reply? action. The parameters url and file explicitly state the correspondence between 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.

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. It is important to observe that the adaptor specification defined by a mapping makes abstraction of 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 Sect. 4. For instance, the behaviour of an adaptor A1 satisfying the specification given by the above mapping M1 is:


This adaptor will keep 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 groups of actions of both components to be related. For instance, consider two components P2 and Q2 involved in an authentication procedure. Suppose that P2 authenticates itself by first sending its user name and then a password. Instead, Q2 is ready to accept both data in a single action:

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 with 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 correspondence.** Adaptation
must also deal with situations in which for
certain actions of one component there is no cor-
responding action in the other one. For instance,
consider a component P3 that features a print-
ing 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 actu-
ally printing the document. Their interfaces are,
respectively:

```
role P3 = { signature printn(Data doc, n);
            behaviour (...) }
role Q3 = { signature setCopies(Data n);
            printn(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 correspond-
ence in P3. The keyword `none` is used to ex-
plicitly represent this asymmetry between com-
ponents.

Notice that in this example the situation is
different from that described for multiple action
consequences. Indeed, the mapping M3 does
not indicate whether Q3 will set the number of
copies for each printing request, or whether a sin-
gle `setCopies!` action will be issued for print-
ing a given number of copies of several docu-
ments. However, a correct adaptor would be
developed in either situation, depending on the
actual behaviours of the two components (delib-
erately omitted in the example), which will be
used for generating the adaptor, as we shall see
in Sect. 4. Notice also that one could enforce the
number of copies to be set for each printing re-
quest by specifying the mapping:

```
M3² = { printn!(doc, n) <!> setCopies!(n),
        print!(doc); }
```

Indeed, M3² specifies a multiple action corre-
sondance 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.

**Nondeterministic 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 manage many possible combinations of ac-
tions independently performed by the two com-
ponents. In order to feature a high-level style of the
specification of the desired adaptation, we allow
nondeterminism in the adaptor specification.

For instance, consider a component P4 sending
a file by means of repeated `data!` actions. Sup-
pose also that the corresponding reader com-
ponent Q4 receives the file with `read?` input ac-
tions, but it may also decide to interrupt the trans-
mission 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 n);
            break!();
            behaviour read?!(n), 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 then be in
charge of building an adaptor capable of dealing
with all the possible specified situations. Once
more, our goal is to allow the adaptor specifi-
cation to ignore many implementation details,
and to leave the burden of dealing with these
details to the (automatic) adaptor construction
process. The use of nondeterministic action cor-
respondences will be further illustrated in Sect. 5.

**Name passing.** The special characteristics of
mobility which are present in the π-calculus al-
low 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 \textit{a priori} as in other process algebras or in object-oriented environments, but can instead be extended by link-passing.

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

\begin{verbatim}
role P6 = {
    signature request!(Data url, Link reply) >
        reply?(Data page);
    behaviour (reply) request!(url.reply).
        reply?(page). 0 }

role Q6 = {
    signature query?(Data handle, Link ret) >
        ret!(Data file);
    behaviour query?(handle,ret). ret!(file). 0}
\end{verbatim}

Here, the situation is slightly different from that of \texttt{P1} and \texttt{Q1}. Role \texttt{P6} indicates that initially the component presents an interface consisting only of the action \texttt{request!}. However, after performing this action, the interface is enlarged with a new link name \texttt{reply}, which must also be considered part of it. This fact is indicated in the signature interface by using the operator `\textgreater` (read as “before”) which explicitly represents the causal dependency between the parameter sent in the action \texttt{request} with the link used later for receiving the reply. Symmetrically for \texttt{Q6}, the link name received as the parameter \texttt{ret} in the \texttt{query?} input action will be used later for sending the return value. The mapping for connecting both components will be:

\begin{verbatim}
M6 = {request!(url.reply)\textgreater query?(url.reply);
    reply?(file) \textless reply!(file); }
\end{verbatim}

4. Adaptor derivation

In the previous section, we have presented a simple notation for expressing a high-level specifica-

tion of the adaptation needed to allow two mismatching components to interact correctly. Given such a specification (mapping) \textit{M}, and the interaction patterns \texttt{P} and \texttt{Q} of two components, a concrete adaptor (if any) will be generated by means of a fully automated procedure. Roughly speaking, such an adaptor will be a component-in-the-middle \texttt{A} such that:

1. The parallel composition \texttt{P}\texttt{A}\texttt{Q} will not deadlock, and
2. \texttt{A} will satisfy all the action correspondences and data dependencies specified by \textit{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 [9] for checking the “so-far correctness” of open contexts of components. Given two patterns \texttt{P} and \texttt{Q}, the algorithm in [9] returns a completion process \texttt{A} (if any) such that the parallel composition \texttt{P}\texttt{A}\texttt{Q} will not deadlock.

To achieve (1), the algorithm tries to build incrementally a completion \texttt{A} by progressively eliminating all the deadlocks that may occur in the evolutions of \texttt{P}\texttt{A}\texttt{Q}. Because of its inherent nondeterministic nature, the construction has been naturally implemented in Prolog.

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

(a) If \texttt{P}\texttt{A}\texttt{Q} may deadlock after executing action \texttt{last}, then an action capable of triggering one of those deadlocked states is nondeterministically chosen (if any), and used
to extend 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\(^2\), 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 extend 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 \( P = a!(\cdot, \tau.b!(\cdot, 0+\tau.a!()\cdot, 0) \) and let \( Q = 0 \) for simplicity. The completion is initially empty and the parallel composition \( P|Q \) is deadlock. Case (a) applies, and action \( a?() \) can be chosen to trigger the context, hence yielding the partial completion \( A = a?(\cdot, 0 \) and setting last to \( a?() \). The new context \( P|(a?(), 0) \) 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 nondeterministically chooses \( b?() \), hence extending the completion into \( A = a?(\cdot, b?(\cdot, 0 \) and setting new last to \( b?() \) the new last. The new context \( P|(a?(), b?(), 0) \) 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

\[ A = a?(\cdot, b?(\cdot, 0.c?()\cdot, 0) \]

\( ^2 \)Indeed in the latter case any attempt to extend \( A \) with a new action “after” last would spoil those successes — see example below.

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 \( P = (a!()\cdot, b!(\cdot, 0+\tau.a!()\cdot, 0) \) and let again \( Q = 0 \) for simplicity. It is easy to see that the only possible trigger \( a?() \) introduces both a successful and a deadlock 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 \( \| \) has to be specialised so as to take into account the constraints specified by the mapping. Basically this amounts to constrain suitably 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 \( \pi \)-calculus process, in which 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 \( M \). Its first rule indicated a one-to-one correspondence between actions \( \text{request!} \) and \( \text{query?} \). Formally, this property can be represented by the following process:

\[ R1 = \text{request?}(\text{url}). (\text{query!}(\text{url}). 0 || R1) + \tau.a.0 \]

stating that whenever the adaptor performs a \( \text{request?} \) input action, then it will eventually have to perform a corresponding \( \text{query!} \) output action. Moreover, according to the data dependencies induced by parameters in the mapping rule, the adaptor should not perform an output action \( \text{query!}(\text{url}) \) until the requested \( \text{url} \) 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 input/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 requirements 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 the 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 respects, the example shows only the relevant details, while hopefully remaining realistic.

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 (thus allowing 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!();

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:

\[
M_A = \{ \begin{array}{l}
\text{login!(usr).} \quad \text{open?(ctl).} \\
\text{pass!(pin) \leftrightarrow user!(usr, pin, ctl);} \\
\text{getfile!(file) \leftrightarrow get?(file, ctl);} \\
\text{logout!() \leftrightarrow close?(ctl);} \\
\end{array} \}
\]

The first rule of \( M_A \) 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 Sect. 4, once an action in 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:

\[
M_B = \{ \begin{array}{l}
\text{open!(ctl).} \\
\text{pass!(pin). (ctl)open!(ctl).} \\
\text{user!(usr, pin, ctl). getfile?(file).} \\
\text{get!(file, ctl). logout!(). close!(ctl);} \\
\end{array} \}
\]

Notice that, even if the actual order 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 \( M_A \).

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.

\[
\text{role IGetDaemon =} \{ \\
\text{interface ctl!(Link data, Link eof) \to} \\
\text{data!(Data x), eof!(());} \\
\text{behaviour (data, eof) ctl!(data, eof).} \\
\text{(tau: data!(x), 0 + tau. eof!(()); 0)} \\
\text{\} the mapping for adapting both roles will be:} \\
\]

\[
M_B = \{ \begin{array}{l}
\text{none \to} \text{ctl!(data, eof);} \\
\text{read?(x) \to} \text{data!(x);} \\
\text{read?(EOF) \to} \text{eof!();} \\
\text{break!() \to} \text{data!(y);} \\
\text{break!() \to} \text{eof!();} \\
\end{array} \}
\]

Its first rule establishes that server action ctl! does not have a correspondence 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 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 ecf! 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, ecf). Once this action is chosen, we have four actions in which the roles are deadlocked: data!, ecf!, read?, and break!. Suppose that the first selected for matching is ecf! the file is empty, yielding the adaptor:

\[ \text{AB} = \text{ctl?(data, ecf). eof?()}.0 \]

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

\[ \text{AB} = \text{ctl?(data, ecf). eof?()}.\text{read!(EOF)}.0 \]

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

\[ \text{AB} = \text{ctl?(data, ecf). eof?()}.(\text{read!(EOF)}.0 + \text{break?()}.0) \]

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:

\[ \text{ctl?(data, ecf)}.(\text{eof?()}.(\text{read!(EOF)}.0 + \text{break?()}.0) + \text{data?(x)}.(\text{read!(x)}.0 + \text{break?()}.0) + \text{break?()}.(\text{data?(x)}.0 + \text{eof?()}.0)) \]

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 is in the well-established research stream which advocates the application of formal methods to describe the interactive behaviour of software systems. More specifically, we adopt the approach of enriching component interfaces with behavioural description for facilitating system analysis and verification in general, [10, 12], and behavioural mismatching detection in particular [13, 15], to cite but a few of the more closely related works. A distinguishing 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 concerning the compatibility of component protocols.

Several proposals for extending IDLs with behavioural aspects are based on finite state machines, like, for instance, [6,11,16]. 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 [10,12,13,17], 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 [18] and CORBA [19], and architecture description languages like Darwin [20] and LEDA [8].

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) [14] and over time (finite interaction patterns) [9], 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 [21–23]). Besides, the problem of software adaptation was specifically addressed by the work of Yellin and Strom [6], 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 to their approach are related to 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 necessary the 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 [24], where software composition is addressed in the context of category theory. The connection between components is obtained 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 byIDL signature descriptions.

As it results from a comparison with significant related works in the literature, and from the representative set of examples shown in this paper, our approach facilitates the adaption of 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 adaption, 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

8. C. Canal, E. Pimentel, and J.M. Troya. Spec-