Electronic Notes in Theoretical Computer Science 66 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume66.html 17 pages

Systematic component adaptation

Andrea Bracciali [1]

Dipartimento di Informatica
Università di Pisa
Italy

Antonio Brogi [2]

Dipartimento di Informatica
Università di Pisa
Italy

Carlos Canal [3]

Departamento de 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 behaviour. The four 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, (3) a fully automated procedure to derive concrete adaptors from given high-level specifications, and (4) an effective technique for verifying properties of adaptors.

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

1 Introduction

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering (CBSE) [4,16,14]. 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 [3].

Available component-oriented platforms (e.g., CORBA [25], COM [7], JavaBeans [26], VisualStudio .NET [20]) address software interoperability typically 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 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 [27], that is, because of differences in the behavior of the components involved. While case-based testing can be performed to check the compatibility of software 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 [9]. 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 the application under development.

In this paper, we focus on the problem of adapting components that may exhibit mismatching behaviour. 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 an application builder has to adapt (manually) a third-party component for using it in a (possibly radically) different context (e.g., see [13,11,17]). A formal foundation for component adaptation was set by Yellin and Strom in their seminal paper [28]. They used finite state machines for specifying component behaviours, and formally 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 four main aspects of the methodology are the following:

(i) *Component interfaces.* We extend traditional IDL interfaces with a description of the behaviour of the components. A component interface therefore consists of two parts: A signature definition (describing the functionality 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 [22] — a process algebra which has proved to be particularly well suited for the specification of dynamic and evolving systems.

(ii) *Adaptor specification.* We present a simple notation for expressing the 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 of the notation is that it produces a high-level, partial specification of the adaptor.

(iii) *Adaptor derivation.* A concrete adaptor component is then automatically generated, given its partial specification and the interfaces of two components. This fully automated process tries exhaustively 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, thus simplifying the task of the (human) software developer.

(iv) *Adaptor properties.* An important part of the described methodology is the formal specification and the verification of properties that an adaptor should satisfy. We show how the constraints defined by an adaptor specification, as well as other interesting properties of adaptors, can be naturally expressed by a set of processes characterising the behaviour of a valid adaptor. An effective way of verifying whether a concrete adaptor satisfies a set of properties is also illustrated.

The rest of the paper is organised as follows. Extended component interfaces are introduced in Sect. 2, while the notation for adaptor specifications is described in Sect. 3. Sect. 4 sketches the automatic process of adaptor generation from specifications. Sect. 5 shows how to express and verify properties of adaptors. Finally, Sect. 6 is devoted to discuss related work and to draw some concluding remarks.

2 Component interfaces

Interfaces will be described in terms of *roles* [5]. 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 resembles traditional IDL descriptions (e.g. CORBA). 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.

The behaviour interface is described by means of what we call an *interaction pattern* [2]. 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 sugared subset of the synchronous π-calculus. Since the calculus permits link names to be sent and received as values, it has proved to be a very expressive notation for describing the behaviour of software components in applications with changing interconnection topology (as those that live in open systems). In particular, we use a polyadic version of π-calculus [21], in which tuples, and not only single names, can be sent along links.

The set of behaviour expressions is formally defined as follows:

`E ::= 0` | `a. E` | `(x)E` | $[x = y]$ `E` | `E || E` | `E + E`

`a ::= tau` | `x?(d)` | `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 non-deterministic choice (+) and parallel (||) operators are defined. The summation `E + E'` may proceed either to `E` or to `E'`. On the other hand, synchronization will only be allowed between expressions belonging to different

components. Hence, `E || E'` consists of expressions `E` and `E'` acting in parallel but not synchronising. Nevertheless, the full π-calculus has also a parallel operator (|) that allows synchronisation, but we do not use it for describing interaction patterns.

Notice that interaction patterns do not contain recursion. The reason is that 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 component `Reader` that sequentially reads a file. File items will be received via 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!()`. The behaviour of the component expressed in full (recursive) π-calculus would be:

```
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 `P1` representing this particular component will simply read:

```
P1 = read?(x). 0 + tau. break!(). 0
```

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 by 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 of software components, 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 conventional programming languages. While type checking cannot in general guarantee the correctness of a program, it does eliminate the vast majority of programming errors. Similarly, while the compatibility of a set of interaction patterns does not guarantee the correctness of a concurrent system, it can eliminate many errors during system assembly [2].

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 would be:

```
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 `P1` previously defined), and the other describing its interaction with the file system, represented by the pattern:

```
P2 = tau. fwrite!(data). 0 + tau. fclose!(). 0
```

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 `P2` succeeds to express the point of view of the file system, for which the reader component seems to decide freely to send either 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 for describing the intended *mapping* among the functionalities of two components being 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 server `Q` that first receives the `url` and then returns the corresponding file. Their behaviour interfaces are, respectively:

```
P = request!(url). reply?(page). 0
```

```
Q = query?(address). return!(file). 0
```

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

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

that permits the interoperation of both components. However, after applying the substitution, the communication between `P`σ and `Q`σ 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. In general, one is interested in adapt-

ing 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 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 parameters `url` and `file` in the mapping explicitly state 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 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 the next section. For instance, the behaviour interface of an adaptor `A` satisfying the specification given by mapping `M` is:

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

This adaptor will maintain the name spaces of `P` and `Q` separated and prevent the two from interacting one another 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 to 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 to specify two important examples of adaptation.

- **Multiple action correspondence.** While the previous example dealt with one-to-one correspondences between the actions in the components, 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:

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

The required adaptation can be specified by the mapping:

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

which associates both output actions performed by `P` to the single input action performed by `Q`. The mapping also shows the use of parameters (viz., `me` and `pwd`) to specify the needed data remembering 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 (action `usr!`), looks for the list of files which are present in a repository (`dir?`), and then deletes two of them (`del!`). On the other hand, the repository server `Q` does not require a login phase, it provides the list of files (`ls!`), but requires the identification of the client for deleting (`rm?`):

  ```
  P = user!(id). dir?(list). del!(f1). del(f2). 0
  Q = ls!(d). rm?(file,usr). rm?(file,usr). 0
  ```

  From the viewpoint of `Q`, authentication concerns are limited to delete actions (`rm`). 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 correspondent in the other component can be associated with `none`. Hence, the following mapping states that the identification phase of `P` has not correspondence in `Q` and also that the parameter `id` must be recorded for subsequent uses.

  ```
  M = { user!(id)  <> none;
        dir?(d)    <> ls!(d);
        del!(f)    <> rm?(f,id); }
  ```

4 Adaptor derivation

In this section we sketch how a concrete adaptor for two roles can be automatically generated, starting from their protocols `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 [2]. Intuitively speaking, the goal of the algorithm is to build a process `A` such that:

(i) `P|A|Q` is successful (i.e. all traces lead to success), and

(ii) `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 deadlocked 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 according to 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. Otherwise it non-deterministically returns one of the adaptors found.

Considering again the last example of Sect. 3 regarding the file repository server, the algorithm may construct, for instance, the following adaptor `A`:

```
A = user?(id). ls?(d). dir!(d). del?(f1). rm!(f1,id).
    del?(f2). rm!(f2,id). 0
```

It is easy to observe that the composition `P|A|Q` is deadlock free, and that `A` satisfies the "intended meaning" of the mapping, both in terms of action correspondence and data dependencies. (For instance, `A` forwards to `Q` the value `id` only after receiving it from `P`.)

It is high time to provide a precise characterisation of what is the "intended meaning" of a mapping, in order to be able to formally state whether an adaptor satisfies or not a mapping. This is the scope of the following section.

5 Verifying properties of adaptors

In this section we show how to formalise and verify whether an adaptor satisfies the intended meaning of a mapping as well as additional requirements.

5.1 Semantics of mappings

As described in Sect. 3, mapping rules establish correspondences between actions in the two components being adapted. Consider again the mapping:

```
M = { user!(id)  <> none;
      dir?(d)    <> ls!(d);
      del!(f)    <> rm?(f,id); }
```

For instance, its second rule indicates how to "translate" the command for listing directories from DOS to Unix syntax. The rule states that *each time* an action `ls!(d)` is performed by one of the components, the adaptor must eventually synchronise with an action `dir?(d)` of the other component. Hence, any adaptor satisfying the rule must suitably match pairs of actions `ls!(d)` and `dir?(d)`.

For representing the semantics of mapping rules we use *properties*, which describe the constraints that the mapping imposes on the adaptor, independently of the actual protocols of the components being adapted. Each property is expressed as a π-calculus process, which performs the actions in the rule an arbitrary number of times. 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, the second mapping rule of the mapping `M` above is represented by the property:

```
R2 = ( ls?(d). dir!(d). 0 || R2 ) + tau. 0
```

i.e., a process that either terminates or performs the actions `ls?(d)` and then `dir!(d)`, in parallel with a recursive instance of itself.
According to the data dependencies induced by parameters in the mapping rule, property `R2` establishes that the adaptor should not perform an output action `dir!(d)` until the list of files `d` is read using an action `ls?(d)`. Property `R2` also states that if the adaptor performs a `ls?(d)` action then it will have to eventually perform a `dir!(d)` action.
Similarly, for the third rule of `M` we have:

```
R3 = ( del?(f). rm!(f,id) || R3 ) + tau. 0
```

However, property `R3` does not express all data dependencies in the third rule. Indeed, in order to perform action `rm!(f,id)`, the adaptor must use a value `id` *previously* read in action `user?(id)` (which appears in the mapping in a different rule). Hence, it is necessary to add a new property to reflect this implicit dependency:

```
R4 = ( user?(id). R4a || R4 ) + tau. 0
R4a = rm!(f,id). R4a + tau. 0
```

Notice how this last property is diverse from the previous properties `R2` and `R3`. Since actions `user?(id)` and `rm!(f,id)` come from different mapping rules, there is no one-to-one correspondence between then, but it still holds

that at least an action `user?(id)` —reading the user identification— must be done before performing (an arbitrary number of times) action `rm!(f,id)`.

Finally, to complete the example, let us consider the first rule of `M`, which maps action `user?(id)` to `none`. Since the rule does not establish any dependency between actions or data, we simply derive the process:

```
R1 = ( user?(id). 0 || R1 ) + tau. 0
```

which states that the adaptor may perform action `user?(id)` an arbitrary number of times.

5.2 Validation of properties

As we have seen, each rule in the mapping gives place to one or more π-calculus processes, representing properties that the adaptor must satisfy. In order to verify them, we take one property at a time, and verify whether the traces performed by the adaptor are "included" in those of the property. Hence, the meaning of the mapping is decomposed into simple, "orthogonal" properties that are checked separately.

Each property refers to actions that occur in the mapping rules from which it was derived, while an adaptor will typically contain also other actions (occurring in other rules of the mapping). Hence, for validating a property we *project* the adaptor over the actions which are relevant to the property being considered. This is done by means of a hiding operator '/' which transforms any action not relevant to the property into an internal `tau` action.

Consider again the example of the file server described in Sections 3 and 4. For the adaptor:

```
A = user?(id). ls?(d). dir!(d). del?(f1). rm!(f1,id).
del?(f2). rm!(f2,id). 0
```

and the properties `R1, R2, R3, R4` we have, respectively:

```
A/R1 = user?(id). tau. tau. tau. tau. tau. tau. 0
A/R2 = tau. ls?(d). dir!(d). tau. tau. tau. tau. 0
A/R3 = tau. tau. tau. del?(f1). rm!(f1). del?(f2). rm!(f2). 0
A/R4 = user?(id). tau. tau. tau. rm!(f1,id). tau. rm!(f2,id). 0
```

In order to check if the traces of the adaptor (once projected) are included in those of the property, inclusion is defined in terms of weak simulation.

Given two π-calculus processes `A` and `R`, `A` is *included* in `R`, written $\texttt{A} \subseteq \texttt{R}$, iff

1) if $\texttt{A} \Longrightarrow \xrightarrow{a} \texttt{A'}$ then $\texttt{R} \Longrightarrow \xrightarrow{a} \texttt{R'}$ and $\texttt{A'}\sigma \subseteq \texttt{R'}\sigma$, for every substitution σ.
2) if $\texttt{A} \Longrightarrow \texttt{0}$ then $\texttt{R} \Longrightarrow \texttt{0}$.

(where $\Longrightarrow$ stands for zero or more `tau` actions).

The definition of inclusion establishes that for `A` being included in `R`, first `R` must be able to perform any action `a` that `A` may perform (preceded in

both cases by a certain number of silent actions); and second that if A may terminate, by reaching inaction, R must be able to terminate, too.

Finally, we say that an adaptor A satisfies a property R iff

$$(\mathtt{A}/\mathtt{R}) \subseteq \mathtt{R}.$$

It is important to observe that the process of property verification always terminates. Indeed, since behaviours are described by finite (non-recursive) interaction patterns, the direct exploration of the traces of the behaviour (A/R) drives a finite exploration of the traces of the recursive process R. For instance, referring back to the previous example, it is easy to check that the adaptor A satisfies properties R1, R2, R3 and R4.

5.3 Expression of additional properties

In the previous sections we have shown how to express formally the meaning of a mapping as a (finite) set of properties, and how to effectively verify whether an adaptor satisfies a set of properties.

The availability of a precise characterisation of the meaning of a mapping has allowed us to formally establish the soundness of the algorithm described in Sect 4.

On the other hand, the possibility of expressing and verifying properties of adaptors permits to validate adaptors which were not produced by the algorithm, as well as to validate adaptors with respect to additional properties.

The properties obtained from the mapping ensure the adequate correspondences between actions in the adaptor. Indeed, this can be an interesting capability of an adaptor, namely to be able of acting as a temporal buffer between two components following different protocols. However, one may wish to check, for instance, whether all the actions in a rule are performed before using again the same rule.

Consider, for instance, two components P and Q exchanging a sequence of two numbers. Suppose that their corresponding protocols are:

```
P = write!(n1). write!(n2). 0
Q = read?(m1). read?(m2). 0
```

Since the actions are differently named in the two protocols, we put an adaptor between them, specified by the mapping:

```
M = { write!(n) <> read?(n); }
```

from which the following adaptor property is derived:

```
R1 = ( write?(n). read!(n). 0 || R1 ) + tau. 0
```

However, it is worth observing that, because of the asynchronous nature of the adaptor, data may not be received in the correct order, as in the case of the following adaptor:

```
A = write?(n1). write?(n2). read!(n2). read!(n1). 0
```

which satisfies R. To ensure that data will be exchanged in the correct order, we can require a given adaptor to satisfy the following additional property:

```
R2 = ( write?(n). read!(n). R2 ) + tau. 0
```

Another interesting property that one may want to specify is that certain data values should be used only once by the adaptor. Consider for instance an editing service where customers must pay in order to print or to spell-check their documents:

```
role EditingService = {
    signature print?(Data doc, Data money);
              spell?(Data doc, Data money);...  }
```

Consider also a customer component which has been designed so as to send payments separately from print and spell-checking requests:

```
role Cliente = {
    signature pagar!(Data dinero);
              imprimir!(Data fichero);
              corregir!(Data fichero); ...  }
```

The required adaptor can be specified by the mapping:

```
M = { pagar!(m) <> none;
      imprimir!(d) <> print?(d,m);
      corregir!(d) <> spell?(d,m); }
```

where the use of the parameter m states the dependencies between actions in the three mapping rules. As shown in previous examples, the properties that will be derived from these rules will ensure that to send a print or spell request to the editing service, the adaptor must first receive a payment from the customer, but this does not prevent the *same* payment from being used for more than one request. To ensure that each different request will be preceded by a different payment, one can enforce valid adaptors to satisfy the following additional property:

```
R = (pagar?(m). (print!(d,m). 0 + spell!(d,m). 0) || R) + tau. 0
```

More generally, one may wish to verify also whether a given adaptor satisfies certain requirements of one of the components. Consider for instance a component that encapsulates a variable to support temporary data storage by offering the methods set? and get!. Consider also a second component that wants to use the service offered by first one according to the mapping:

```
M = { write!(x) <> set?(x);
      read?(y) <> get!(y); }
```

One may wish to check that, after the introduction of the adaptor, the second component is used in a coherent way, not reading the variable before it has been initialised. This can be enforced by introducing the additional

property:

```
R = set?(x). Ra + tau. 0
Ra = set?(x). Ra + get!(x). Ra + tau. 0
```

To sum up, in this section we have shown how to specify different properties on adaptors, and how these properties are can be checked once the adaptor is developed. We are currently working on taking them into account in the derivation algorithm itself, so they can be validated on-the-fly during the construction of the adaptor.

6 Concluding remarks

Several authors have proposed to extend current IDLs in order to deal with behavioural aspects of component interfaces. The use of finite state machines to describe the behaviour of software components is proposed for instance in [8,19,28]. The main advantage of finite state machines 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 [1,23,24], and support system simulation and formal derivation of safety and liveness properties. In particular the usefulness of π-calculus has been illustrated for describing component models like COM [12] and CORBA [15], and architecture description languages like Darwin [18] and LEDA [5]. 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) [6] and over time (finite interaction patterns) [2].

A general discussion of the issues of component interconnection, mismatch and adaptation is reported in [11,13], while formal approaches to detecting interaction mismatches are presented for instance in [1,6,10]. The problem of software adaptation was specifically addressed by the work of Yellin and Strom [28], 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 internal 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.

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 [2,6], 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.

After laying a foundation for a systematic development of adaptors, we intend to devote our future activities to develop a user-friendly environment to facilitate experimenting the proposed methodology on real CBSE applications.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–49, July 1997.

[2] A. Bracciali, A. Brogi, and F. Turini. Coordinating interaction patterns. In *ACM Symposium on Applied Computing (SAC'2001)*. ACM Press, 2001.

[3] A.W. Brown and H.C. Wallnau. The current state of CBSE. *IEEE Software*, 1998.

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

[5] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Software Architecture*, pages 107–126. Kluwer Academic Publishers, 1999.

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

[7] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[8] 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.

[9] 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-Verlag, 1994.

[10] D. Compare, P. Inverardi, and A. L. Wolf. Uncovering architectural mismatch in component behavior. *Science of Computer Programming*, 33(2):101–131, February 1999.

[11] 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 Verlag, 1997.

[12] L.M.G. Feijs. Modelling Microsof COM using π-calculus. In *Formal Methods'99*, number 1709 in LNCS, pages 1343–1363. Springer Verlag, 1999.

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

[14] D. Garlan and B. Schmerl. Component-based software engineering in pervasive computing environments. In *4th ICSE Workshop on Component-Based Software Engineering*, 2001.

[15] 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.

[16] George T. Heineman. An evaluation of component adaptation techniques. In *2nd ICSE Workshop on Component-Based Software Engineering*, 1999.

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

[18] 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 Verlag, 1995.

[19] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Software Architecture*, pages 35–49. Kluwer Academic Publishers, 1999.

[20] Microsoft Corporation. .NET Programming the Web. http://msdn.microsoft.com.

[21] R. Milner. The polyadic π-calculus: a tutorial. Technical report, University of Edinburgh, Octubre 1991.

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

[23] 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 Verlag, 1999.

[24] 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 Academic Publishers, 1999.

[25] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group. http://www.omg.org.

[26] Sun Microsystems. JavaBeans API specification. http://java.sun.com/beans/docs.

[27] 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 Verlag, 2000.

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