Dynamically adapting the behaviour
of software components

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

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

Abstract. Available component-oriented platforms address software interoperability only at the signature level, while they do not provide suitable mechanisms for adapting components with mismatching interaction behaviour. This paper presents a methodology for automatically developing adaptors capable of solving behaviour mismatches between heterogeneous components. These adaptors are generated from abstract specifications of the intended connection between the components, by taking into account both signature interfaces and component behaviours.

1   Introduction

The ability to reuse existing software has always been a major concern of software engineering. In particular, the so-called component-based software development focusses on reusing and integrating heterogeneous software parts, as currently supported by several component-oriented platforms like CORBA, COM, JavaBeans, or .NET.

A serious limitation of the available platforms is that they do not provide suitable means to describe and reason on the concurrent behaviour of interacting components [14]. Indeed, while these platforms provide convenient ways to describe signatures via interface description languages (IDLs), they offer a quite limited and low-level support to describe the concurrent behaviour of components.

Several proposals [10] have been put forward to enhance component interfaces with a description of their concurrent behaviour [1–3, 12]. Indeed, the availability of a formal description of component behaviour is the basis for verifying properties of systems consisting of large numbers of heterogeneous components. For instance, one may wish to check whether two components are compatible, that is, whether their interaction may lead or not to a deadlock.

Our objective is to address the problem of dynamically constructing adaptors capable of overcoming existing mismatches between heterogeneous components that may be separately developed. We will illustrate a methodology to derive an adaptor starting from the protocols of the components involved and from a high-level partial specification of their intended connection. The three main aspects of the methodology are the following:

- *Component interfaces.* Each component has a *signature* interface — in the style of traditional IDLs — which specifies the methods it provides and requires, together with type information on their parameters. Components also have a *behaviour* interface which describes their actual behaviour, that is, the order in which the methods in the signature interface are invoked. Behaviour interfaces are expressed in a subset of π-calculus [13], a process algebra which has proved to be specially suited for the specification of dynamic and evolving systems.
- *Adaptor specification.* A high-level specification of an adaptor is expressed by means of a simple notation indicating the adaptation needed for the interoperation of two components with mismatching behaviour. The specification simply states some correspondences between actions and parameters of the two components, abstracting from many behavioural concerns.
- *Adaptor derivation.* Finally, an automatic procedure deploys, if possible, the abstract description into an adaptor which lets the components interoperate in terms of their behaviour protocols. The deployed adaptor ensures successful execution, while keeping disjoint the name spaces of the components so that their interaction will occur only through the adaptor.

Our work is motivated by the ever-increasing attention devoted to developing extensively interacting distributed systems, consisting of large numbers of heterogeneous components. Most importantly, we think that constructing adaptors dynamically will be a must for the next generation of nomadic applications consisting of wireless mobile computing devices that will need to require services from different hosts at different times. The following scenario concretely relates our work to this perspective:

1. A component P gets in the vicinity of a context C of interacting components. P gets from C its signature interface, describing the services that C provides;
2. Then, P sends to C its interaction protocol, together with a proposal of connection in the form of a *mapping* between the interface of C and its own.
3. The context C, given this connection proposal, the protocol of P, and its own protocol, constructs an adaptor to be used for their interoperation.

In this paper we will focus on the notation to be used in step (2) for specifying the mapping, and on the adaptor construction procedure of step (3). The (semantical) problem of defining the mapping for two given components is at present under strong investigation, (e.g. the use of XML as "Universal Data Fromat" in the .NET framework), but it is out of the scope for this paper.

The mapping in step (2) is only a partial specification of the intended connection. It focusses on the mediation between the different languages spoken by P and C. Thus, it refers to signature interfaces, abstracting from the actual protocols of P and C represented by their behaviour interfaces. On the contrary, in step (3) protocols are considered in order to develop automatically an adaptor satisfying both the mapping and the behaviour interfaces.

In the rest of this paper, Sect. 2 shows how component behaviour can be described by means of finite interaction patterns. Sect. 3 introduces a notation

for the specification of mappings, while Sect. 4 presents an algorithm for the generation of adaptors. Finally, Sect. 5 discusses related works and some concluding remarks.

2 Interface description

In our approach, the interface of a software element (either a single component or a full context) is described by a set of *roles*, each one devoted to a specific aspect of the behaviour of the component.

Roles consist of both a signature and a behaviour interface description. The signature declares a set of input and output actions, representing the methods that the component offers and invokes, the values or exceptions returned, etc.

With respect to the behaviour, it is specified by a finite interaction pattern. The language we use for describing these patterns is a sugared subset of the synchronous π-calculus. Since the calculus allows link names to be sent and received as values, it has proved to be a very expressive notation for describing software applications with changing topologies (as open systems are). The set of behaviour expressions is defined as follows:

```
E ::= 0  |  a. E  |  (x)E  |  E || E'  |  E + E'

a ::= x?(d)  |  x!(d)  |  tau
```

where `0` represents inaction, input and output actions are respectively represented by `x?(d)` and `x!(d)`, and internal actions are denoted by `tau`. Restrictions — like `(x)E` — represent the creation of a new link name `x` in an expression `E`. The parallel (not synchronizing) operator is represented by '‖' inside roles, while the operator '|' is used for general parallel composition. Finally, '.' and '+' denote the standard prefix and choice operators.

Notice that behaviour expressions do not contain recursion, since they are intended to specify finite fragments of interaction. Intuitively speaking, an interaction pattern describes only the essential aspects of the interactive behaviour that a component may (repeatedly) show to its environment.

The special characteristics of mobility of the π-calculus allow the creation and transmission of link names which can be later used for communication. This determines that the interfaces of π-calculus processes are not fixed beforehand (like in other process algebras or in object-oriented environments), but instead they can be extended by link-passing. Consider, for instance, the pattern `C` below representing a server waiting for some query:

```
C = query?(quest,ret). ( tau. ret!(fact). 0 + tau. exception!(). 0 )
```

Initially, the signature of `C` consists of actions `query?` and `exception!`, link names which are public in the role. However, after performing the `query?` action, the interface is enlarged with an output action `ret!` to be performed through the link name received in action `query?`, and which was not known beforehand. We use the operator '>' (read as "before") for representing these changing interfaces,

stating causal dependencies between action names without showing the details of the corresponding protocols. Hence the signature interface of `C` will be written as:

```
I_C = { query?(Data quest, Link ret) > ret!(Data fact);
        exception!(); }
```

3   Connection specification

The intended connection between a component and a context is represented by a mapping between the signature interfaces of both elements. The specification is partial in the sense that mappings do not deal with behavioural concerns but they rather address problems like name translation and parameter reordering between two signature interfaces.

A mapping is a finite set of rules of the form $\alpha_1, \ldots, \alpha_m <> \beta_1, \ldots, \beta_n$. The left term refers to component actions, and the right one to context actions. Each term may contain one or several actions. The names of action parameters are used to describe reordering, remembering, and even synthesis of links and data during the adaptation process, indicating how data would be transmitted from one element to the other. Moreover, a rule may map some actions to `none`, indicating that there is no corresponding action for them. When executed, these actions will be "absorbed" by the adaptor when necessary, but not retransmitted.

For instance, consider a component `Q` that sends requests for printing a number of copies of a document by means of an action `printc!(doc,n)`, and another component `R` that features a printing service by accepting requests in two steps —one for setting the number of copies, and one for actually printing the document. Their signature interfaces are, respectively:

```
I_Q = { printc!(Data doc, Data n); }    I_R = { setCopies?(Data copies);
                                                 print?(Data doc); }
```

A suitable mapping for connecting `Q` and `R` is:

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

which indicates that the action `setCopies?` in `R` does not have a correspondent in `Q`. The example also shows that the scope of parameter names is not just a rule, but the whole mapping, and that any value received by the adaptor may be used later for matching an action in a different rule.

Notice also that mappings define matchings between actions, as represented in the signature interfaces. In the example above, the mapping does not indicate whether `R` requires to set the number of copies for each printing request, or whether an initial set is enough for printing a given number of copies of several documents. However, a correct adaptor would be developed in both situations, according to the actual behaviour of the two components (not shown here), which are used for generating the adaptor.

Consider now a component `P` described by the role:

```
I_P = { request!(Data q);        P = request!(q).reply?(a).0
        reply?(Data a); }
```

In order to connect P to the context C described in the previous section, we will
write the mapping:

```
M = { request!(q)      <> query?(q,reply);
      reply?(fact)     <> reply!(fact);
      reply?(REFUSED)  <> exception!() }
```

The first rule resolves name mismatching, mapping `request!` to `query?`. More-
over it implicitly states that the `reply` channel used by P corresponds to the
return link that C expects for its parameter `ret` in action `query?`. In fact, when
dealing with mobility we must take into account that links appearing in the sig-
nature interface after the operator '>' refer to parameters which are instantiated
in the mapping rules. Hence, further reference to these links must be renamed
consequently, as shown in the second rule, where the action `ret!` of C is referred
to as `reply!`. Finally, the third rule states that the context action `exception!` is
also mapped to `reply?`. Notice how a constant value `REFUSAL` is used, allowing
P to distinguish between real answers and service refusals.

4 Adaptors

Given two roles P and C and a mapping M that specifies the intended connections
between them, the construction of a mediating component, called *adaptor*, can be
done automatically. Because of space limitations, in this section we only sketch
the algorithm and show an example of its application. Intuitively, the goal of the
algorithm is to build a process A such that:

1. P|A|C is successful (i.e., all traces lead to success), and
2. A satisfies the given mapping M, that is, all the action correspondences and
   data dependencies specified by M are respected in any trace of P|A|C.

The algorithm incrementally builds the adaptor A by trying to eliminate pro-
gressively all the possible deadlocks that may occur in the evolution of P|A|C,
letting P and C interoperate according to M. Informally, a deadlock is removed
by expanding A with an action α by means of which the adaptor synchronises
with either P or C. The algorithm is non-deterministic, since action α is non-
deterministically chosen among those that P or C are ready to synchronise with.
   The algorithm returns, if any, one of the possible adaptors for P and C. Notice
that all the adaptors returned by the algorithm are correct (P|A|C is successful)
and satisfy M. It is worth noting that the algorithm is a variant of the one we
developed for checking the correctness of open contexts of components [2].
   In order to produce adaptors that satisfy the given mapping, the algorithm
traces the evolution of P|A|C by means of the following data structures:

 – The set $\mathcal{D}$ of data acquired by matching output actions of P or C. It is used
   to enforce that the adaptor sends data only after receiving them.

- The set $\mathcal{F}$ of actions to be eventually matched by the adaptor. If an action occurring in a mapping rule is matched, all the other actions in the same rule must be eventually matched, and they are hence recorded in $\mathcal{F}$.
- The set $\mathcal{L}$ of link correspondences is used to guarantee the separation of name spaces between P and C.

The construction of the adaptor is a iterative process whcih starts from an initially inactive adaptor. At each iteration, the algorithm:

- Chooses an action α so as to match a dual action $\overline{\alpha}$ on which P or C is blocked. Note that not all actions $\overline{\alpha}$ can be matched by the adaptor, since it must respect the constraints of M.
- For each choosable α, the algorithm spawns an instance of itself, relative to A appropriately expanded with α. If there are no triggerable actions, then the instance *fails*.

Each instance terminates when the derivation tree of P|A|C does not contain deadlocks. If the set $\mathcal{F}$ of actions to be matched is empty, then the instance terminates successfully, and it returns the completed adaptor. Otherwise, it fails. The overall algorithm fails if all its instances fail. Otherwise, it non-deterministically returns one of the adaptors found.

Let us consider again the client/server system whose intended connection was specified in Sect. 3. Starting form A = 0, the system P|A|C is clearly deadlocked on actions request!(q) and query?(quest,ret), respectively of P and C. Of these actions, A can only match the first one, since it is not able at the moment to provide the data required for the second. Hence, the adaptor is expanded to A = request?(q). 0. As a consequence of this first step, $\mathcal{D} = \{q\}$ is updated with the data received by A, while, according to the first rule of M, the action query!(q,reply) is stored in the set $\mathcal{F}$ of actions to be eventually fired.

After A and P have synchronised on request, there is only one deadlock state that A can eliminate, matching query?(quest,ret) of C with query!(q,reply) which is taken from the set $\mathcal{F}$. However A must take care of keeping separate the link names of the two roles. In order to do that, A creates a new link, say reply', for sending it to C. The link relation reply-reply' is stored in $\mathcal{L}$, while A is expanded to:

    A = request?(q). (reply') query!(q,reply'). 0

and action query!(q,reply) is removed from $\mathcal{F}$ (which now is empty), since it has been executed.

P|A|C offers now two triggerable actions: reply'!(s) and exception!(), and A can match both of them. Two (eventually successful) instances of the algorithm are spawned. Following the first one, the construction continues by extending A with the actions reply'?(s) and reply!(s). Observe how the relation between reply and reply' stored in $\mathcal{L}$ is used while applying the second rule of the mapping. On the other hand, the second instance of the algorithm extends A with actions exception?() and reply!(REFUSED), by applying the

third rule in the mapping. In both cases the roles of `P` and `C` terminate. Summing up, `A` has been extended to:

```
A = request?(q). (reply') query!(q,reply').
      ( reply'?(s). reply!(s). 0 + exception?(). reply!(REFUSED). 0  )
```

It is easy to verify that now `P|A|C` does not contain deadlocks, but only two successful traces. The set $\mathcal{F}$ is empty, and hence the algorithm terminates successfully returning `A`. Notice that the name spaces of `P` and `C` are still separated, and hence they are not able to interact without using the adaptor. This ensures that their connection will respect the specification represented by the mapping.

5 Concluding remarks

Several authors have proposed to extend current IDLs in order to deal with behavioural aspects of component interfaces. Some proposals use finite state machines to describe the behaviour of components, e.g [12, 15], exploiting their simplicity in supporting an efficient verification of protocol compatibility. On the other hand, this simplicity is a severe expressiveness bound for modelling complex open distributed systems.

Process algebras feature more expressive descriptions and analysis of concurrent systems, and support system simulation and formal derivation of safety and liveness properties [1]. In particular the usefulness of π-calculus has been illustrated for describing component models like COM [7] and CORBA [9], and architecture description languages like Darwin [11] and LEDA [4]. However, the main drawback of using process algebras is related to the inherent complexity of the analysis. In order to manage this complexity, previous work of the authors has described the use of modular and partial specifications, by projecting behaviour both over space (roles) [4] and over time (finite interaction patterns) [2]. In the present work we use a combination of both approaches.

A general discussion of the issues of component interconnection, mismatch and adaptation is reported in [6, 8], while formal approaches to detecting interaction mismatches are presented for instance in [1, 4, 5]. The problem of software adaptation was specifically addressed by the work of Yellin and Strom [15], which constitutes the starting point for our work. They use finite state grammars to specify component behaviour, to define a relation of compatibility, and to (semi)automatically generate adaptors. Some significant limitations of their work are related with the expressiveness of the notation used, which, for instance, does not allow the representation of internal choices, parallel compositions or process creation. Furthermore, the architecture described is static, and does not deal with issues such as reorganizing the communication topology of systems. 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 results presented in this paper lay a foundation for the automatic development of adaptors capable of solving behavioural mismatches between heterogeneous interacting components. An interesting direction for future work is

the formal verification of properties of the generated adaptor, such as security properties, as suggested in [12, 15]. In practice, such a verification would allow a context to check that its invariant properties are preserved after accepting a new component, and it would also allow the joining component to check that its minimal connection requirements will be fullfilled by the proposed adaptor. Our future activities will also be devoted to develop a user-friendly environment to facilitate experimenting the proposed methodology on real CBSE applications. In this perspective, we are planning to optimize the performance of the algorithm described in Sect. 4 by applying finite model-checking techniques. We are also investigating a similarity-based hierarchy of adaptors so as to fully automize the selection of the adaptor returned by the algorithm.

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. C. Canal et al. Extending corba interfaces with protocols. *Computer Journal*, 44(5):448–462, 2001.
4. 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.
5. D. Compare, P. Inverardi, and A. L. Wolf. Uncovering architectural mismatch in component behavior. *Science of Computer Programming*, 33(2):101–131, 1999.
6. 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.
7. L. M. G. Feijs. Modelling Microsof COM using π-calculus. In *Formal Methods'99*, number 1709 in LNCS, pages 1343–1363. Springer Verlag, 1999.
8. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.
9. M. Gaspari and G. Zavattaro. A process algebraic specification of the new asynchronous CORBA messaging service. In *Proceedings of ECOOP99*, number 1628 in LNCS, pages 495–518. Springer, 1999.
10. G. T. Leavens and M. Staraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
11. 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.
12. J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Software Architecture*, pages 35–49. Kluwer Academic Pub., 1999.
13. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.
14. 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.
15. 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.