On the dynamic adaptation of component behaviour

Carlos Canal

Dept. 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. The previous work of the author has addressed the development of a formal methodology for adapting at run-time components with mismatching interaction behaviour. In this position paper, I present a brief overview of the methodology proposed, show its relations with some other significant works in the field, and discuss some interesting open issues that deserve further research work.

1 Introduction

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering (CBSE) [6, 16, 14]. The possibility of adapting existing software components to work properly within new applications is a must for the creation of a true component marketplace and for component deployment in general [5]. Indeed, the ability to reuse existing software has always been a major concern of Software Engineering. In particular, component-based software development —focused on reusing and integrating heterogeneous software parts—, is partially supported by current component-oriented platforms like CORBA, J2EE, or .NET. These platforms address several adaptation issues, allowing some degree of interoperability between software components, even when they are built by independent third-parties possibly using heterogeneous programming languages.

Interoperability can be defined as the ability of two or more entities to communicate and cooperate despite differences in the implementation language, the execution environment, or the model abstraction [25]. Basically, three main levels of interoperability can be distinguished: *(i)* the *signature level*, dealing with names and signatures of operations; *(ii)* the *behavioural level*, dealing with protocols (i.e. the relative ordering of the messages exchanged), and *(iii)* the *semantic level*, dealing with the functionality offered by an entity, (i.e. the "meaning" of its operations) [24].

Interoperability is currently well-defined and understood at the signature level, for which middleware architects and vendors are trying to establish different interoperational standards —such as the platforms mentioned above—, and bridges among them. However, a serious limitation of these platforms is that

they do not provide suitable means to describe and reason on the concurrent behaviour of interacting components [7]. Indeed, while current component 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. Consequently, all parties are starting to recognize that this sort of signature interoperability is not sufficient for ensuring the correct development of large applications in open systems.

In fact, component interoperability should be studied in general at the semantic level. However, this is quite an ambitious and broad problem, very difficult to tackle in full. As a first step, the *state-of-the-art* of the research in this field concentrates on the interoperability of reusable components at the behavioural level, where the basic problems can be identified and managed, and where practical solutions can be proposed to solve them. As a result, several proposals have been put forward to enhance component interfaces with a description of their concurrent behaviour (see, for instance [1, 7, 20]). 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.

2 Previous work

The previous work of the author in this field[1] has addressed the problem of dynamically constructing adaptors capable of overcoming existing mismatches between heterogeneous components that may be separately developed [3, 4]. This work has lead to the development of a methodology to derive automatically adaptors starting from the description of the behavior of the components involved, and from a partial and high-level specification of the connection intended between them. The three main aspects of the methodology proposed are the following:

- *Component interfaces.* Each component provides both a *signature* interface —in the style of traditional IDLs, describing the methods it provides and requires, together with type information on their parameters—, and also a *behaviour* interface, which describes its actual behaviour, that is, the order in which the methods in the signature interface are invoked. Behaviour interfaces are expressed using a notation based on standard process algebras.
- *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 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 a concrete adaptor component which lets the components being adapted interoperate in terms of their behaviour protocols.

[1] These works have been done in collaboration with Andrea Bracciali and Antonio Brogi form the University of Pisa, and Ernesto Pimentel and Antonio Vallecillo from the University of Málaga.

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.

This work was 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 pervasive applications running on wireless mobile computing devices that will require services from different hosts at different times.

3 A scenario for dynamic component adaptation

To illustrate the problem of adapting mismatching interaction behaviour, let us consider the following over-simplified example. Suppose that a component `PrinterClient` has been designed so that it sends printing requests by assuming that the service provider will offer two separate services: one for setting the number of copies, and one for printing a document. The interaction pattern followed by `PrinterClient` with respect to the printer may be formalised by the process algebra term:

```
PrinterClient = setCopies!(n). print!(doc). 0
```

Suppose now that the above component needs to be integrated with another component `Printer` that features a printing service by directly accepting requests to print a number of copies of a given document. This behaviour of the `Printer` is expressed by the term:

```
Printer = printc?(doc,copies). Printer
```

Clearly, the interaction patterns above need to be suitably adapted one to another in order for the two components to effectively interoperate. A natural solution is to introduce an adaptor component between them. The adaptor can be specified by a mapping `M` between actions in the signature interface of the two components:

```
M = { setCopies!(n) <> ;
      print!(doc)   <> printc?(doc,n) }
```

Taking this specification, and the behaviour interfaces of the components being adapted, the `Adaptor` component can be automatically deployed. In this simple example, the adaptor must collect the two requests sent by the `PrinterClient` so as to re-arrange them in the format accepted by the `Printer`:

```
Adaptor = setCopies?(x). print?(y). printc!(y,x). 0
```

In spite of the simplicity of this example, protocol adaptation is in general a difficult problem to be faced. Indeed, the interaction protocols of the components involved may mismatch in different ways. Adaptation could range from the simple translation of message names and the re-arrangement of parameters, to a more radical reordering and synthesis of messages and data, even leading to the inhibition of whole parts of a component protocol.

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.

The mapping in step (2) is only a partial specification of the intended connection. It focuses 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.

4 Related Work

The work of the author in the field of component coordination and adaptation 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 already mentioned, several authors have proposed to extend current IDLs in order to deal with behavioural aspects of component interfaces. The use of finite state machines (FSM) to describe the behaviour of software components is proposed for instance in [9, 20, 26]. The main advantage of FSM is that their simplicity allows an efficient verification of protocol compatibility. On the other hand, this same 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, 22], and support system simulation and formal derivation of safety and liveness properties. In particular, the π-calculus —differently from FSM and other algebras like CCS— is able to model some relevant features for component-based open systems, like local and global choices, dynamic creation of new processes, and dynamic reorganization of network topology. The usefulness of π-calculus has been illustrated for describing component models like COM [12] and CORBA [15], and architecture description languages like Darwin [19].

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, behaviour interfaces have to be described in an abstract and modular way. Modularity is achieved by the partition of the interface in several *roles* each one describing the a partial view of component behaviour, as seen from a particular partner involved in the interaction. Abstraction, can be provided by the use of *behavioural types* —such as session types [17]—, instead of full process algebras. The ultimate objective of employing behavioural types is to provide a basic means to describe complex interaction behaviour with clarity and discipline at a high-level of abstraction, together with a formal basis for analysis and verification. Behavioural types are supported by a rigorous type discipline, thus featuring a powerful type checking mechanism of component behaviour. Moreover, the use of types —instead of processes— to represent behaviour features the possibility of describing recursive behaviour while maintaining the analysis tractable.

A general discussion of the issues of component interconnection, mismatch and adaptation is reported in [2, 11, 13], while formal approaches to detecting interaction mismatches are presented for instance in [1, 8, 10]. The problem of software adaptation was specifically addressed by the work of Yellin and Strom [26], 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.

The outstanding paper of Yellin and Strom is probably one of the starting works in this field. However, 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 more expressive foundations, like session-types or the π-calculus.

Another closely related work is that of Reussner [23], who proposes the extension of interfaces with FSM in order to check correct composition and also to adapt non-compatible components. Protocols are divided into two views: the services the component offers, and those it requires from its environment. In their proposal, these two views are orthogonal, i.e. each time a service is invoked in a component it results the same sequence of external invocations, though this usually depends on the internal state of the component. It should be also noticed that only method invocation is considered, while in a more general setting other forms of interaction should be addressed. Finally, adaptation is considered in this work as *restriction* of behaviour; if the environment does not offer all the resources required, the component is restricted to offer a subset of its services, but no other forms of adaptation (like name translation, or treatment of protocol mismatch) is considered.

Also similar in goals and approach is the work of Inverardi and Tivoli [18], who address the automatic synthesis of connectors in COM/DCOM environments. Their approach assumes a layered system architecture in which the con-

nectors/adaptors play the role of data buses carrying and translating messages between components located in adjacent layers. The formal approach used in their proposal guarantees deadlock-free interactions between components. However, the capability of adaptation achieved is limited to the (immediate) translation of message names between component interfaces, and for instance their connectors cannot act as buffers, temporarily storing messages to be transmitted later, and therefore adapting bigger behavioural mismatch between the components involved.

5 Open Issues

As shown, there is a big research effort being put in the field of automatic and dynamic software adaptation, going further the mere signature adaptation provided by currently available commercial component platforms, and addressing the more complex behavioural adaptation. However, there are also some interesting issues still open, deserving future research.

First of all, many of the works in the literature of component adaptation —among them author's own works—, deal with adapting behaviour (i.e. the protocols that the components follow in their interactions), rather than functionality (i.e. the actual semantics of the computations associated to these interactions). However, solving all pending issues in the behavioural level will allow components to interact successfully, but cannot ensure at all the correctness of the system. In fact, behavioural specifications are deprived from the semantic information of the messages exchanged —that is, the functionality actually carried by a component when the operation corresponding to a received message is invoked. Hence, behavioural adaptation is useless if the mapping between interface specifications is wrong or meaningless.

Rigorous description of component functionality can be achieved by means of *contracts* [21], using pre- and post-conditions for describing the semantics of component's services. Hence, we would knew beforehand whether we can use a given component within a certain context. However, in case that semantic mismatch is detected, some issues arise: *(i)* under which circumstances would it be possible to adapt the functionality that a component offers to that required by its context? *(ii)* would these contract descriptions be enough for deriving a "semantic adaptor"? and *(iii)* if not, what other kind of information should be included in the interfaces in order to achieve automatic adaptation?

Furthermore, if our goal is to perform adaptation in a highly dynamic environment —such as in a scenario of pervasive computing—, the specification of the adaptation required (i.e. what in our proposal we call the mapping between the interfaces of the components being adapted) must be automatically generated at run-time. Hence, a `PrinterClient` running on a wireless device must be aware that it is entering the scope of a `Printer`, but it should also be able to determine which is the *meaning* of the different services offered by the printer (i.e. which printer operation actually prints a document). The (semantic) problem of defining the mapping for two given components is at present under strong

investigation, (e.g. the use of XML-based notations as a kind of "Universal Data Format"), but it is out of the scope of most of the proposals mentioned above. The works currently being carried in the context of the *Semantic Web* could help to do that, by defining ontologies of services from which derive adaptor specifications.

Finally, there are also other important issues in adaptation, apart from functionality; mismatch of non-functional properties should be addressed too. There is no point in adapting a `PrinterClient` to a given `Printer`, if the printer is too slow for client needs, or if it cannot print documents with the quality required. However, if extending component interfaces for describing in full functionality is a hard task, specifying (and then adapting) non-functional requirements seems to be currently out of reach; it would involve the use of different notations for specifying each property, and also different adaptation machinery for solving each kind of mismatch.

As shown, automatic adaptation of component requires to extend interfaces in order to describe all the services provided by a component, and also the assumptions it makes of the outside world. Two other interesting issues here are *(i)* who is going to provide these heavy-weighted interfaces? (specially for legacy components) *(ii)* is there a way of generating them automatically from the component implementation? and *(iii)* how could we ensure that a component actually behaves/corresponds to what is stated in its interface? (the latter being related to some other interesting issues such as security and service payment, among others).

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–49, 1997.
2. J. Bosch. Adapting object-oriented components. In *2nd. International Workshop on Component-Oriented Programming (WCOP'97)*, pages 13–22. Turku Centre for Computer Science, Sept. 1997.
3. A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software, Special Issue on Automated Component-Based Software Engineering*, 2004. (In press). A preliminary version of this paper was published in *Component Deployment*, LNCS 2370, pages 185–199. Springer, 2002.
4. A. Brogi, C. Canal, and E. Pimentel. Soft component adaptation. Electronic Notes in Theoretical Computer Science (ENTCS), 85(3), 2003.
5. A. Brown and H. Wallnau. The current state of CBSE. *IEEE Software*, 1998.
6. G. H. Campbell. Adaptable components. In *ICSE 1999*, pages 685 – 686. IEEE Press, 1999.
7. C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Adding roles to CORBA objects. *IEEE Transactions on Software Engineering*, 29(3):242–260, March 2003.
8. C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41:105–138, 2001.
9. I. Cho, J. McGregor, and L. Krause. A protocol-based approach to specifying interoperability between objects. In *Proceedings of TOOLS'26*, pages 84–96. IEEE Press, 1998.

10. D. Compare, P. Inverardi, and A. L. Wolf. Uncovering architectural mismatch in component behavior. *Science of Computer Programming*, 33(2):101–131, 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, 1997.
12. L. Feijs. Modelling Microsof COM using π-calculus. In *Formal Methods'99*, number 1709 in LNCS, pages 1343–1363. Springer, 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. G. T. Heineman. An evaluation of component adaptation techniques. In *2nd ICSE Workshop on Component-Based Software Engineering*, 1999.
17. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *European Symposium on Programming (ESOP'98)*, number 1381 in LNCS, pages 122–138. Springer, 1998.
18. P. Inverardi and M. Tivoli. Automatic synthesis of deadlock free connectors for COM/DCOM applications. In *Procs. of ESEC/FSE'2001*. ACM Press, 2001.
19. J. Magee, S. Eisenbach, and J. Kramer. Modeling darwin in the π-calculus. In *Theory and Practice in Distributed Systems*, number 938 in LNCS, pages 133–152. Springer, 1995.
20. J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Software Architecture*, pages 35–49. Kluwer, 1999.
21. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
22. E. Najm, A. Nimour, and J. Stefani. Infinite types for distributed objects interfaces. In *Proceedings of the third IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'99*. Kluwer, 1999.
23. R. H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *34th Hawaiin International Conference on System Sciences*. IEEE Press, 2001.
24. A. Vallecillo, J. Hernández, and J. M. Troya. New issues in object interoperability. In *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, number 1964 in LNCS, pages 256–269. Springer, 2000.
25. P. Wegner. Interoperability. *ACM Comp. Surveys*, 28(1):285–287, March 1996.
26. D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.