Systematic Component Adaptation

Antonio Brogi

Department of Computer Science
University of Pisa, Italy
[brogi@di.unipi.it]

Abstract. Component adaptation is widely recognised as one of crucial problems in component-based software development. In this talk, we will present a formal methodology for adapting components with mismatching interaction behaviours. The four main ingredients of the methodology are:

1. The inclusion of behaviour specifications in component interfaces,
2. a simple, high-level notation to express adaptor specifications,
3. a fully automated procedure to derive concrete adaptors from given high-level specifications, and
4. an effective technique to verify properties of adaptors.

1 Motivations

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering (CBSE) [4]. 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 [2].

Available component-oriented platforms (e.g., CORBA, COM, JavaBeans, VisualStudio .NET) address software interoperability 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 [7], that is, because of differences in the behaviour 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 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 (manually) adapt a third-party component for a (possibly radically) different

use (e.g., see [3, 5]). A formal foundation for component adaptation was set by Yellin and Strom in their seminal paper [8]. 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.

2 A systematic approach to component adaptation

In this talk, I will first present a formal methodology for adapting components with mismatching interaction behaviours. Following [1], the four main aspects of the methodology are the following:

- *Component interfaces.* Traditional IDL interfaces are extended with a description of component behaviour. 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 [6] — a process algebra which has proved to be particularly well suited for the specification of dynamic and evolving systems.
- *Adaptor specification.* A simple notation is introduced to express 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 the notation is to allow a high-level, partial specification of the adaptor.
- *Adaptor derivation.* A concrete adaptor component is then automatically generated, given its partial specification and the interfaces of two components. The process of adaptor generation consists of exhaustively trying 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.
- *Adaptor properties.* An important part of the described methodology is the formal specification and the verification of properties that an adaptor should satisfy. Indeed the constraints defined by an adaptor specification, as well as other interesting properties of adaptors, can be naturally expressed and verified by means of processes characterising the behaviour of a valid adaptor.

3 Concluding remarks

The above described methodology employs π-calculus [6] to describe the interaction behaviour of components. Indeed π-calculus has proved to be particularly

well suited for the specification of concurrent applications with changing topologies, such as those that live in open systems.

In the last part of this talk, I will try to illustrate how computational logic — and logic programming in particular — can be exploited for the process of deriving a concrete adaptor from a given high-level specification. Indeed the Prolog programming language is particularly well-suited to program the nondeterministic incremental construction of a possible adaptor capable of letting two components properly interact according to a given specification.

An interesting direction for future work is to investigate the use of computational logic to fully automatize the adaptation of components. More precisely, the adaptor specification is currently generated by the user after analyzing and comparing the interfaces of two components. The symbolic reasoning capabilities of computational logic may sensibly contribute to automatize the analysis of component interfaces represented in RDF/XML formats.

Acknowledgements. I would like to thank A. Bracciali and C. Canal for their contribution in developing the methodology presented in this talk. This research was partly supported by the EU-funded project SOCS IST-2001-32530.

References

1. A. Bracciali A. Brogi and C. Canal. Adapting components with mismatching behaviours. In J. Bishop, editor, *Component deployment, IFIP/ACM Working Conference*, LNCS 2370, pages 185–199. Springer-Verlag, 2002.
2. A.W. Brown and H.C. Wallnau. The current state of CBSE. *IEEE Software*, 1998.
3. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.
4. G. T. Heineman. An evaluation of component adaptation techniques. In *2nd ICSE Workshop on Component-Based Software Engineering*, 1999.
5. S. Hissam K. Wallnau and R. Seacord. *Building Systems from Commercial Components*. The SEI Series in Software Engineering, 2001.
6. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.
7. 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.
8. 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.