Dynamic adaptation using contextual environments

Javier Camara, Carlos Canal, Javier Cubo, Ernesto Pimentel

Dept. of Computer Science, University of Málaga (Spain)
{jcamara,canal,cubo,ernesto}@lcc.uma.es

Abstract. By dynamic adaptation, we mean the ability to modify an application at run-time. This provides a system with the skill to dynamically alter its behaviour while it is running, depending on the (changing) conditions of the environment. In this work we show how to perform dynamic adaptation by means of contextual environments, which define flexible adaptation policies. Thus, our main goal is to describe a context-dependent mapping between the interfaces of the components being adapted, as opposed to static mappings presented in previous works. We present a case study in order to illustrate the proposal. We also discuss the improvements that our current proposal represents in comparison with previous works, as well as some open issues.

1 Introduction

Software Adaptation (SA) is a key issue for the development of a real market of components for the advance of software reuse. The main aim of Software Adaptation is to enhance the flexibility and maintainability of systems [8]. The old notion of developing a system by writing code has been replaced here by assembling existing components. Thus, in an ideal scenario, component-based systems would be built from pre-produced Commercial-Off-The-Shelf (COTS) components, by plugging together perfectly compatible components, which in conjunction achieve the desired functionality. However, it turns out that the constituent components often do not fit one another when they are going to be reused, and adaptation has to be done to eliminate the resulting mismatches [2]. Therefore, the software composition always requires a certain degree of adaptation [8,13], and its purpose is to ensure that conflicts among components are minimised.

Here, we consider the problem of adapting mismatching behaviour that components may exhibit. The notion of adaptor was introduced formally in [16], being defined as a software entity capable of enabling components with mismatching behaviour to interoperate. Component-oriented platforms like CORBA, J2EE or .NET address several adaptation issues, allowing a certain degree of interoperability between software components. Indeed, they provide convenient ways to describe signatures using Interface Description Languages (IDLs), but they offer a quite limited and low-level support to describe the concurrent behaviour of components, since solving all signature problems does not guarantee that the components will suitably interoperate. In fact, mismatch may also occur at the protocol level, due to the ordering of exchanged messages, and also to blocking conditions [14], that is, because of behavioural mismatch of the (possibly) heterogeneous software components involved. Furthermore, component interoperability should be studied in general at the semantic level, but this is quite an ambitious and broad problem, very difficult to tackle in full.

As a first target, recent research effort [1,7,12] concentrates on the interoperability of reusable components at the behavioural level, since the basis for the verification of system properties consisting on two or more heterogeneous components is a well-defined formal description of component behaviour.

Our current proposal focuses in defining flexible adaptation policies by means of contextual environments, providing a system with the ability to dynamically alter its behaviour during execution depending on the (changing) conditions of the environment. This is a broader scenario than that presented in our previous works [4,5], where the mapping was static or immutable. Hence, it could not control the dynamically changing conditions of the system. It is interesting mentioning, that by dynamic adaptation, we mean the ability to change a specification at run-time.

The structure of this paper is the following: In Section 2, we briefly present the module calculus [3] we use in our proposal, and we draw a formal notation that defines the proposal. Section 3 presents a case study in order to illustrate our approach, indicating the improvements in comparison with previous works. Finally, Section 4 draws up the main conclusions of this paper and sketches some future tasks that will be accomplished to extend its results.

2 Overview of the proposal

Some recent works in the field of Software Adaptation have addressed several problems related to signature and behavioural mismatch. In this section, we outline an approach for dynamic software component adaptation.

The first step needed to overcome behavioural mismatching is to let behaviour information be explicitly represented in component interfaces. Typing component behaviour and service specification applied in recent works [1,6,9,10,11] have been described both in terms the process algebra and of session types, with their uses, advantages and drawbacks discussed in works as [5,15]. A suitable formalism to express adaptor specifications is also required. The desired adaptation will be expressed by simply defining a set of (possibly non-deterministic) correspondences between the actions provided by the two (or more) components to be adapted.

A limitation of the adaptation technique described in [4] is that it is somewhat rigid, in that it only succeeds if there is an adaptor that strictly satisfies the given specification. Indeed, in many situations an adaptor could be nevertheless deployed by weakening some of the requirements stated in the specification. Hence, we extended the aforementioned methodology precisely to overcome this limitation and presented the results in [5,6]. The idea was featuring a secure, *soft* adaptation of third-party software components when the given adaptation requirements could not be fully satisfied. Technically this was achieved by exploiting the notion of *subservice* (substitution of a service for another one which features only a limited part of its functionality) to suitably weaken the initial specification when needed. Correspondingly, component interfaces are extended with a declaration of their subservice relations as well as with the *access rights* needed to access the component services. But a pending question in that approach was how to deal with access rights that may change dynamically.

2.1 A module calculus for dynamic adaptation

In this proposal, we briefly describe context-dependent flexible mappings in order to solve the problem of dynamic component adaptation, overcoming the limitations of static mappings presented in [4,5,6]. Indeed, this work aims to achieve a richer expressiveness and flexible mappings in contrast with the reduced expressiveness of immutable mappings from previous works. For that purpose, we make use of a module calculus presented in [3]. The main goal is to obtain a dynamic mapping through the use of contextual environments.

The module calculus in [3] defines a small set of operators over environments and modules, designed to express various encapsulation policies, composition rules, and extensibility mechanisms. Using these operators, it is possible to specify a set of module combinators (composing and manipulating modules) that capture the semantics of modules in different object-oriented programming languages. This module calculus employs the primitive notion of *environment* (mapping from some domain *D* to an extended range $R^* = R \cup \{\perp\}$), and *modules* are defined as abstractions over environments.

We employ the module calculus in order to define the formalism to express adaptor specifications. Aforementioned, we propose a methodology, briefly presented in the following section, to obtain dynamic mappings depending on the changing conditions of the environment, which will permit to approach the problem of dynamic component adaptation using contextual environments and modules.

2.2 Drawing the proposal

This Section is devoted to outline a brief description of our proposal, which is based in the module calculus presented in [3]. Table 1 shows an informal definition which describes the approach we use in order to obtain a context-dependent dynamic mapping between the interfaces of the components being adapted.

The definition of the set of operators over environments and modules, mentioned in Section 2.1, falls out of the scope of this work, and it is already presented in [3]. We describe below the specifications shown in Table 1. The symbol $\odot$ represents function composition.

Table 1. Adaptor specifications using the module calculus described in [3].

Actions and Agents	
Actions I/O	$a,b,c,\ldots$
Agents	$(P,Q,R,S,\ldots\in)\,Agent$
Environments and Modules	
Environment	$(\varepsilon\in)\,E = 2^D \to 2^R$
Contextual Environment (mapping)	$(\gamma\in)\,\Gamma = 2^D \to 2^R \odot Agent \to \Gamma$ $\gamma\in\Gamma \; if \; \exists((\varepsilon\in E)\wedge(\mu\in Agent)) \to \Gamma \; / \; \gamma(\alpha) = \begin{cases} \varepsilon(\alpha) & if \;\; \alpha\in 2^D \\ \mu(\alpha) & if \;\; \alpha\in Agent \end{cases}$
Module (mapping)	$(m\in)\,M = \Gamma \to \Gamma^*$

In our proposed notation, we suppose behavioural interface of the components will be given by *agents* (processes) specification in some process algebra, and our adaptor specifications will map message names (*actions*) in *D* to *R*, contextually depending on the agent for which they are defined. Correspondences between messages in both components are established. We have different options in order to map these correspondences: a message in one part (component) may have no correspondence in the other part (component); or one or more messages that belong to one of the components may correspond to either only one action or different actions in the other component. Then, it is necessary defining an *environment* as a function mapping 2^D (parts of *D*) to 2^R (parts of *R*), where *D* and *R* are the alphabets used by the components. With the objective of simplifying the notation, when there is a single action (message) mapped (in a component) we will denote it without {}.

We represent *environments* as finite sets of mappings. For example:

$$\varepsilon_1 \; = \; \{a \mapsto 1, b \mapsto 4, c \mapsto \{3,4\}, \{\;\} \mapsto 2\}$$

is an environment that maps *a* to 1, *b* to 4, *c* to 3 and 4, and defines no correspondence to 2. In [3] all the values in the domain of the environment which are not mapped, correspond to *bottom* ($\perp$), however here we add the possibility to specify values without correspondence, which will be mapped to {}.

On the other hand, we represent *contextual environments* as functions taking *2^D* or an *Agent* (which denotes behavioural interface of the components) as domain, and returning *2^R* or a contextual environment as image (note that this responds to recursive definition):

$$\gamma_1 \; = \; \{a \mapsto 1, Q \mapsto \{c \mapsto 2, R \mapsto \{S \mapsto \{a \mapsto 4, c \mapsto \{3,4\}\}\}\}, b \mapsto 4\}$$

is an contextual environment where *Q*, *R* and *S* are agent (process) definitions in the behavioural specification of the components being adapted. This contextual environment maps *a* to 1. In the context of *Q*: *c* is mapped to 2; and in the context of *R*: within the context of *S*, *a* is mapped to 4, and *c* to 3 and 4. Out of those contexts, *b* is mapped to 4.

Finally, we represent *modules* as functions taking a contextual environment and returning a contextual environment:

$$m_1 \; = \; \lambda\gamma.\{a \mapsto 1, Q \mapsto \{c \mapsto 2, R \mapsto \gamma\}, b \mapsto \gamma Qc\}$$

As we see in m_1, the γ parameter makes it possible for entries in a module to look up other bindings in the parameter contextual environment. In this case, the mapping is the following: *a* is mapped to 1 (everywhere); within the context of *Q*, *c* is mapped to 2, and the context of *R* is mapped to the contextual environment γ. Last, *b* is mapped to the result applying within the contextual environment a mapping from *c* in a defined value in the context of *Q* (if the action *c* has no correspondence in the context of *Q* within the contextual environment γ, then *c* will be mapped to {} (*empty*)).

3 Case study and comparison to previous works

We present a simple case study in order to illustrate the methodology of this proposal, and then we discuss the improvements this work purports in comparison with previous ones. The example consists on a simplified *Video-on-Demand* (VoD) system taken from [5], which is a Web service providing access to a database of movies and news. In Table 2, we present the VoD behavioural specification. We assume a typical scenario where a *Client* component wishes to use some of services offered by the *VoD* service. The client will ask for the VoD interface, and then submit its service request in the form of an adaptor specification that is a contextual mapping as described in Section 2.1. For each request, a daemon session (*Daemon* agent) is opened. The daemon is handed over to the client, and the VoD returns to its initial state, allowing concurrent access to the system.

Table 2. Specifications of *VoD* service: system behaviour interface and client interface.

VoD: behaviour specification
$VoD = open?(\ ).\ (suscrip?(priv).\ Daemon \mid VoD)$
$Daemon = search?(title).\ list!(movies).\ Daemon$
$+\ preview?(item).\ stream!(video).\ Daemon$
$+\ view?(item).\ (play?(\ ).\ stream!(video).\ Daemon$
$+ record?(\ ).\ stream!(video).\ Daemon)$
$+\ news?(date).\ stream!(news).\ Daemon$
$+\ suscrip?(priv).\ Daemon$
$+\ close?(\ ).\ 0$
Client: adaptation specification
$Client = user!(priv).\ menu!(\ ).\ info?(list).$
$(watch!(title).\ data?(movie).\ 0$
$+\ store!(title).\ data?(movie).\ 0$
$+ user!(priv).\ 0)$

In the system, there are four different profiles of clients depending on certain access rights. Thus, there exist registered and unregistered (by default) users. The former are those paying a regular fee, and they are divided into *news*, *movies* and *full* clients, while the latter (*guest*) are only allowed to *search* for a movie in the VoD catalogue, *preview* it for a few minutes, and quit the system. Clients with *news* profile have the same capabilities as *guest*, but they may also *watch* and *store* the news (see contextual environments defined in Table 3). The *movies* profile grants access to *view* movies but not the news, while *full* clients may access both news and movies.

We will take the different profiles as the contexts of the system, and they could change through actions *user* (*Client* requests to the system the modification of its privileges) and *suscrip* (*Client* is subscribed to the requested profile). Furthermore, we will take into account the possibility that the services may not be available at some point during the execution.

When a client opens a session with the VoD system (no correspondence in *Client* is mapped to the action *open* in *VoD*), it follows a connection procedure which associates the session with one of the four profiles described (*guest* by default), depending on the identity of the client which will be given by the mapping from *user to suscrip*. The "*priv*" parameter will indicate the user privileges. Then, a *Daemon* session is thrown and the *Client* could perform any permitted action (*watch*, *store*), or this could also dynamically change its access rights, obtaining a new context with a different behavioural interface. For example, once a specific movie has been selected for viewing, *movies* users might start its visualization (*play*), while *full* users might also decide to *record* it permanently in their computers. They are different context environments with distinctive mappings (described in contextual environments of Table 3). Finally and following the sequence, the *Daemon* process finalizes its session (no correspondence necessary for the action *close*).

In our example, the client profile by default is *guest*, thereby in Table 3 the initial adaptation by default is for a *guest* client (see *Daemon* context). But the access rights can change at run-time (actions *user* and *suscrip*), depending on *Client* context (*Movies*, *News*, *Full*), and accordingly it will dynamically change the mapping. Thus, if a client begins its execution with *guest* profile, this will only have the privileges by default, but if, for example, it changes its profile to *movies* client, it will acquire new permissions corresponding to the *Movies* contextual environment (represented in Table 3). In the mapping shown in Table 3, we have a *Daemon* context, where the mapping between both components (*VoD* service and *Client*) is presented. Within the *Daemon* context, in the context of *Client*, a user has *movies* profile, so that the client could change the access rights, or perform any other authorized action to a *movies* client, according to the behavioural interface for the *Movies* contextual environments.

Table 3. Dynamic adaptation with contextual environments (client profiles).

Adaptation: correspondence of actions (and data)
$VoD \;=\; \lambda\gamma.\ \{\{\ \} \mapsto open?(\ ),\ user!\ (priv) \mapsto suscrip?(priv),$
$\quad Daemon \mapsto \{menu!(\ ) \mapsto search?(''\ ''),$
$\quad\quad info?(string) \mapsto list!(string),$
$\quad\quad watch!(title) \mapsto preview?(title),$
$\quad\quad store!(title) \mapsto preview?(title),$
$\quad\quad Client \mapsto \gamma,$
$\quad\quad data?(video) \mapsto stream!(video),$
$\quad\quad \{\ \} \mapsto close?(\ )\}\}$
Contextual environments: client profiles (by default *Guest*)
$Movies = \{user!(priv) \mapsto suscrip?(priv),$
$\quad watch!(title) \mapsto \{view?(title).\ play?(\ )\},$
$\quad store!(title) \mapsto \{view?(title).\ play?(\ )\}\}$
$News = \{user!(priv) \mapsto suscrip?(priv),$
$\quad watch!(title) \mapsto news?(date),$
$\quad store!(title) \mapsto news?(date)\}$
$Full = \{user!(priv) \mapsto suscrip?(priv),$
$\quad watch!(title) \mapsto \{view?(title).\ play?(\ )\},$
$\quad store!(title) \mapsto \{view?(title).\ record?(\ )\},$
$\quad Movies \mapsto \{store!(title) \mapsto \{view?(title).\ play?(\ )\}\},$
$\quad News \mapsto \{watch!(title) \mapsto news?(date),$
$\quad\quad store!(title) \mapsto news?(date)\}\}$

Mapping : *VoD* system and *Movies* (client profile)

$$
\begin{aligned}
VoD \;=\; \lambda\gamma.\ \{\{\ \} \mapsto open?(\ ),\ & user!\ (priv) \mapsto suscrip?(priv),\\
& Daemon \mapsto \{menu!(\ ) \mapsto search?(''\ ''),\\
& \qquad info?(string) \mapsto list!(string),\\
& \qquad watch!(title) \mapsto preview?(title),\\
& \qquad store!(title) \mapsto preview?(title),\\
& \qquad Client \mapsto \{user!(priv) \mapsto suscrip?(priv),\\
& \qquad\qquad watch!(title) \mapsto \{view?(title).\,play?(\ )\},\\
& \qquad\qquad store!(title) \mapsto \{view?(title).\,play?(\ )\}\},\\
& \qquad data?(video) \mapsto stream!(video),\\
& \qquad \{\ \} \mapsto close?(\ )\}\}
\end{aligned}
$$

An improvement of the notation proposed in this paper in comparison with previous works is that this new technique intends not only to get a dynamic adaptation by the fact the system alters its behaviour at run-time, but also because of the adaptation (mapping) changes during the execution of the system, depending on changing conditions of the environment (in this case, depending on client profiles). Therefore we obtain an adaptation in which a message is mapped to different actions, according to the state of the environment (context). However, in [4,5] the mapping was static, so a command was always translated to the same sequence of messages (actions).

4 Conclusions and open issues

We have presented throughout this paper a description of a formal notation for contextual component adaptation. The purpose of this new technique is to obtain dynamic mappings between the interfaces of the components being adapted, through contextual environments that define flexible adaptation policies. In our previous works, this idea was presented, although from a different point of view (*subservice* and *right access*). With this approach, we intend to overcome some of our previous constraints, making a significant advance to find a solution for issues like dynamic access rights (user privileges) by contextual adaptation.

We have employed a module calculus defined in [3] to obtain a high degree of expressiveness through flexible mappings, so that message translation between the components will change depending on the conditions of the system. In order to exemplify our proposal we have presented a case study in this paper relative to a *Video-on-Demand* (VoD) system with different client profiles (contextual environments).

This notation has still certain limitations. It is worth noting that our proposal constitutes a modular and dynamic approach of specifying the required adaptation between two software components. Thus, an interesting extension is to consider adaptation between three or more interoperating components, and composition among them. Likewise, it will be important to take into account the derivative problems of the recursion in this new proposal, but it will be an open issue to deal with in future work. An issue to be studied more profoundly is the way of alteration of the environment conditions for which the mapping changes at run-time.

The distinguishing aspect of the notation used is that it produces a high-level, partial specification of the adaptor required. A specific adaptor component will be generated via a fully automated procedure. The adaptor must guarantee the safe interaction of the adapted components (verification of properties), making sure that they will never deadlock during an interaction session. Furthermore, an interesting future work will be to develop an adaptor generation process, founded on the algorithm presented in [4], producing an adaptor which provides the maximum possible flexibility.

We look forward to contribute in the research on adaptation issues, so that we can continue advancing in this formal technique. Although we will also explore other possible ways to describe highly expressive mappings for solving component mismatch.

References

1. Allen, R., Garlan, D.:A Formal Basis for Achitectural Connection. In *ACM Trans. on Software Enginnerring and Methodology*, 6(3):213-49, ACM Press, 1997.
2. Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M.: Towards an Engineering Approach to Component Adaptation. *Dagstuhl Seminar* 04511: *Architecting Systems with Trustworthy Components*. Springer-Verlang, LNCS 3938, 2006.
3. Bergel, A., Ducasse, S., Nierstrasz, O.: Analyzing Module Diversity. In *Formal Aspects of Component Software* (FACS'2005), *Electronic Notes in Theorical Computer Science* (ENTCS), Elsevier (in press).
4. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45-54, Elsevier, 2005.
5. Brogi, A., Canal, C., Pimentel, E.: Component adaptation through flexible subservicing. *Science of Computer Programming*, Elsevier, 2006 (in press).
6. Brogi, A., Canal, C., Pimentel, E.: On the semantics of software adaptation. *Science of Computer Programming*, Elsevier, 2006 (in press).
7. Canal, C., Fuentes, L., Pimentel, E., Troya, J.M., Vallecillo, A.: Adding roles to CORBA objects. *IEEE Transactions on Software Engineering*, 29(3):242-260, March 2003.
8. Canal, C., Murillo, J.M., Poizat, P.: Software adaptation. *L'Objet, Special Issue on the 1st International Workshop on Coordination and Adaptation of Software Entities* (WCAT'04), vol. 12, num. 1, pp. 9-31. Hermes, 2006.
9. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In *European Symposium on Programming* (ESOP'98), volume 1381 of LNCS, pages 122-138. Springer, 1998.
10. Inverardi, P., Tivoli, M.: Automatic synthesis of deadlock free connectors for COM/DCOM applications. In ESEC/FSE'2001. ACM Press, 2001.
11. Magee, J., Eisenbach, S., Kramer, J.: Modeling darwin in the π-calculus. In *Theory and Practice in Distributed Systems*, LNCS 938, pages 133-152. 1995.
12. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour analysis of software architectures. In *Software Architecture*, pages 35-49, Kluwer, 1999.
13. Nierstrasz, O., Meijler, T.D.: Research Directions in Software Composition, *ACM Computing Surveys*, vol. 27, num. 2, 1995, pp. 262–264.
14. Vallecillo, A., Hernández, J., Troya, J.M.: New issues in object interoperability. In *Object-Oriented Technology*, LNCS 1964, pages 256-269. Springer, 2000.
15. Vallecillo, A., Vasconcelos, V.T., Rabara, A.: Typing the behaviour of objects and components using session types. *Electronics Notes in Theorical Computer Science* (ENTCS), 68(3), 2003.
16. Yellin, D.M., Strom, R.E.: Protocol specifications and components adaptors. In *ACM Transactions on Programming Languages and Systems*, 19(2):292-333, ACM Press, 1997.