An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution

Javier Camara[1], Carlos Canal[1], Javier Cubo[1], Juan Manuel Murillo[2]

[1]Dept. of Computer Science, University of Málaga (Spain)
{jcamara,canal,jcubo}@lcc.uma.es
[2] University of Extremadura (Spain),
[2]Dept. of Computer Science, Quercus Software Engineering Group,
juanmamu@unex.es

Abstract. This paper briefly describes the design of a dynamic adaptation management framework exploiting the concepts provided by Aspect-Oriented Software Development (AOSD) -in particular Aspect-Oriented Programming (AOP)-, as well as reflection and adaptation techniques in order to support and speed up the process of dynamic component evolution by tackling issues related to signature and protocol interoperability. This will provide a first stage to a semi-automatic approach for syntactical and behavioural adaptation.

1 Introduction

One of the most significant trends in the software development area is that of building systems incorporating pre-existing software components, commonly denominated commercial-off-the-shelf (COTS). These are stand-alone products which offer specific functionality needed by larger systems into which they are incorporated. The purpose of using COTS is to lower overall development costs reducing development time by taking advantage of existing and well tested products. But this approach to systems engineering has its drawbacks: development teams have no control over the functionality, performance, and evolution of COTS products because of their Black-Box nature. Moreover, in most of the cases these components are not designed to interoperate with each other, requiring customized adaptation which has to be performed time and again when teams face their integration along the evolution of the system. These activities are highly demanding, consuming time and resources which could otherwise be devoted to the enhancement or development of new functionality.

The need to automate the aforementioned adaptation tasks has driven the development of Software Adaptation (SA) [4], a new discipline characterized by highly dynamic run-time procedures that occur as devices and applications move from network to network, modifying or extending their behaviour. SA promotes the use of software adaptors [12], specific computational entities for solving interoperability problems between software entities which can be classified in four different levels:

Signature Level: Interface descriptions at this level specify the methods or services that an entity either offers or requires. These interfaces provide names, type of arguments and return values, or exception types. This kind of adaptation implies solving syntactical differences in method names, argument ordering and data conversion.
Protocol Level: Interfaces at this level specify the protocol describing the interactive behaviour that a component follows, and also the behaviour that it expects from its environment. Indeed, mismatch may also occur at this protocol level, because of the ordering of exchanged messages and of blocking conditions. The kind of problems that we can address at this level is, for instance, compatibility of behaviour, that is, whether the components may deadlock or not when combined.
Service Level: This level groups other sources of mismatch related with non-functional properties like temporal requirements, security, etc.
Semantic Level: This level describes what the component actually does. Even if two components present perfectly matching signature interfaces, and they also follow compatible protocols, we have to ensure that the components are going to behave as expected.

We will focus in the design of a framework based on Software Adaptation techniques and how they can be applied in order to support and speed up the process of Software Evolution, particularly at the signature and protocol levels. Considering the aforementioned opaque nature of COTS components, the techniques provided for the development of this framework should be non-intrusive. In this sense, AOP [6] makes a perfect candidate, providing a mechanism to extend and modify the behaviour of components without directly altering them (i.e., their code). We should also

employ automatic and dynamic procedures, in order to enable adaptation just in the moment in which components join the context of the system (or are substituted as the system is running). The development of this kind of framework can provide a new breeding ground for the development of agile methodologies for Software Evolution by reducing integration effort supporting (semi)automatic component adaptation.

In this paper, Section 2 discusses the advantages provided by different approaches to dynamic AO component adaptation, and justifies the convenience of selecting Dynamic Adaptor Management. Although signature level is the state-of-the-art in adaptation (e.g. CORBA's IDL-based signature description), several proposals have been made in order to enhance component interfaces with a description of their concurrent behaviour [1, 3, 7], allowing automatic adaptor derivation in some circumstances [2]. Section 3 briefly describes the design of a dynamic adaptation management framework based on the concept of automatic adaptor derivation and gives some tips on implementation issues using AspectJ. At last, section 4 presents some conclusions and open issues.

2 Supporting Unanticipated Dynamic Software Evolution: Alternative Strategies based upon AO and Adaptation

When performing dynamic component adaptation, it is important to have reliable, transparent and automatic procedures and mechanisms. These often require information only available at runtime. If we want to take advantage of this information, we have to find a way to apply it at runtime as well in order to modify the behaviour of the components. We may consider two different strategies:

> ***Dynamic Aspect Generation***: Adaptors are implemented by means of aspects which are generated, applied and removed at runtime as required. This approach increases the complexity of the infrastructure required for execution, demanding some non-trivial modifications to it, such as the inclusion and integration of new functionality (runtime aspect code generation and compilation). Alas, the computational overhead caused by these additional tasks may be too heavy if the system is not carefully optimized. On the other hand, this approach would provide a high degree of flexibility in adaptor generation.
>
> ***Dynamic Adaptor Management***: Several precompiled aspects manage adaptation. In this approach, the different aspects form a manager which is able to retrieve, interpret, and use the dynamic information required for adaptation. Different adaptors can be built using the algorithm described in [2], and managed specifically for each interaction between components as they join the context of the system and invoke methods belonging to others.

So far, several efforts have been made in the community in order to develop platforms such as CAM/DAOP [10] or PROSE/MIDAS [11], which are already capable of performing dynamic aspect weaving, a mechanism that allows aspect code to be woven into an application at any point of its execution. This technique will enable the application of adapting aspects independently of the selected approach. Although the state of the art does not currently make Dynamic Aspect Generation a feasible approach, it is a promising choice to consider for future research. Dynamic Adaptor Management, on the other hand may be less flexible but suffices the requirements to perform dynamic adaptation, and the required infrastructure in comparison is much simpler. This justifies the adoption of this strategy for this first stage of our proposal.

3 Dynamic Adaptation Management Framework

3.1 System Architecture

When performing dynamic adaptation, we require both signature and protocol information from the components being adapted to produce a consistent *mapping* or correspondence between their interfaces in order to solve potential mismatches. This can either be obtained from the components using techniques for the incorporation of metadata such as annotations [5], or semantic techniques [8] exploiting the already available information from the components, and inferring protocol related information such as order of message exchange in a similar fashion to OWL-S [9], used in the field of Web Services. While in the former approach adaptation may work more accurately, the latter does not require the component to be specifically prepared for adaptation. However, the available information may vary depending on the specific platform we are using, so a compromise may be necessary, such as taking a hybrid approach by adding

some complimentary information to the components in some cases if it is required. Anyway, the construction of such mappings falls out of the scope of this paper, and it is an issue to discuss in itself in further work. For our purposes, we will consider that the mapping is already available, focusing in the design of an aspect-based adaptation management framework. As we can see in Fig.1, the architecture of the system contains three basic functional modules in charge of the different tasks required for adaptation:

Interface Manager: This module is in charge of inspecting the interfaces of the components as they join the context of the system, and keeping their description in an interface repository in order to use them later for mapping generation. For this purpose we will use reflection techniques. Upon initialization of the component c_1 of class *C*, the manager checks for the existence of an entry for *C* in the repository, and if it does not exist, it creates one for it. Each one of these entries is a set of information containing a minimum of method and argument names, argument and return value types, argument ordering, and exception types. As we have already mentioned, this set of information can be extended with other properties (protocol-related, etc.). This may be required at some point in order to solve some specific problems, although it should not be encouraged, since the principle of obliviousness would be compromised.

Adaptor Manager: It generates new adaptors as required by the conditions of the system. Once a component of class *S* joins the context, it may generate one or several messages to other components. Every time one of these messages is generated, the manager captures it and checks if it is the first one consigned to a target component of class *T*. If that is the case, a mapping is produced between the source and target component classes, and subsequently an adaptor is automatically generated making use of the algorithm described in [2]. This adaptor is stored in a repository and it will be used for interaction management between any pair of components of classes (*S*, *T*). This module will incorporate an inference engine based on pre-agreed ontologies explicitly defining resources, preconditions, and effects of processes, as well as domain related properties and relationships. In such a way, we will provide the system with a machine-interpretable description of the semantics of the components. This enables the use of inference techniques traditionally used in AI (knowledge representation, goal-oriented planning, logic, etc.) in order to infer relevant properties from the components and adapt them. Once generated, these adaptors will allow syntactical adaptation providing message and parameter name translation, data conversion, and parameter reordering. They will also provide a mechanism to perform protocol adaptation, storing messages whenever required for a delayed delivery, and establishing correspondences between them which can be one-to-one as well as one-to-many. By accessing the Adaptor Manager the engineers can supervise and tune the behaviour of the components by editing the mappings produced by the inference engine in order to fit specific needs. The characteristics of these mappings may also be constrained by manual introduction of contextual information in the engine. This capability enables a semi-automatic approach in which the engineer can easily evolve components worrying mostly about coarse-grained issues.

Coordination Manager: Monitors and translates all messages between components. Each time a component s_i sends a message to a component t_i, the manager translates it making use of the already available adaptor for (*S*, *T*) stored in the repository. A pool for session information is established in this manager in order to store specific information about the state of the components and their interaction. For each pair of interacting components (s_i, t_i), a session is created in the repository the first time s_i sends a message to t_i. This session information is updated if necessary with each message between components. Session information will be publicly available to the mechanisms in the coordination manager since some interactions between components may influence that of others.

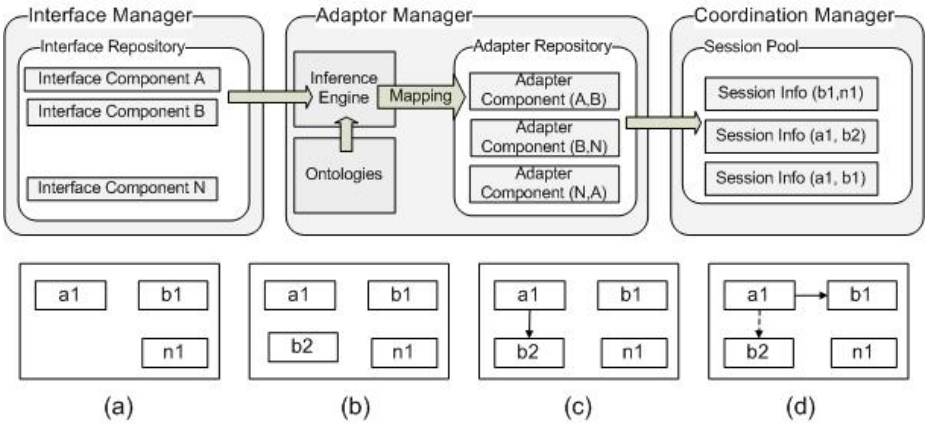

Fig. 1. Architecture diagram and simple component interaction example: Components *a1*,*b1*, and *n1* join the context. Interfaces *A, B,* and *N* are stored in the interface repository **(a)**. Component *b2* joins the context **(b)**. *a1* sends a message to *b2*. Interfaces for *A* and *B* are mapped and adaptor (*A* ,*B*) is generated in the adaptor repository and a session entry for components (*a1*,*b2*) is created in the session pool. The message is then translated by the coordination manager **(c)**.*a1* sends a message to *b1*. A session entry is then created for components (*a1*,*b1*) in the session pool and the message translated by the coordination manager **(d)**.

3.2 Implementation Issues

In order to illustrate some of the issues related to the implementation of our proposal, we will make use of AspectJ, which is highly representative of the AOP systems currently used. In this section we will highlight some of the key structures and mechanisms it provides to implement the functionality of our adaptation management framework. If we take a look at its design, we can enumerate a minimum set of pointcuts we have to define in order to provide the required functionality:

Component initialization: It is satisfied whenever a new component enters the context of the system. It will be used by the interface manager in order to store interface related information.

Component invocation: Specifies all the messages sent from one component to another within the context of the system. Used by the adaptor manager for adaptor generation and by the coordination manager for session creation, message translation, and session info updating.

It is worth mentioning that since multiple aspects are present in the system, pieces of advice in the different aspects corresponding to each of the managers, may apply to a single join point. When this situation is given, the order in which advices are applied to the join point must be explicitly defined. This is the case of component invocation, which is used both by the adaptor and the coordination managers. In order to observe this order, AspectJ uses precedence rules to determine the sequence in which advices are applied. Aspects with higher precedence execute their before advice on a join point before the ones with lower precedence. When the method of a component is invoked, the sequence to follow is: **(a)** the adaptor manager checks if an adaptor needs to be generated. **(b)** The coordination manager checks if a session entry must be created, and **(c)** the coordination manager translates the message and updates session information. This translation is driven by the previously generated mapping and implemented through the join point model provided by AOP. This provides an elegant and non-invasive way of performing message translation.

AspectJ also provides mechanisms for source and target component identification through the use of `thisJoinPoint getThis()` and `getTarget()` methods. The coordination manager can monitor argument values in method invocations making use of the `getArguments()` method provided by `thisJoinPoint` as well. In order to obtain information related to methods such as exception, return, and parameter types, as well as argument and method names we can use the `getSignature()` method provided by `thisJoinPointStaticPart`.

Table 1. Pointcut definition and main API classes used for the framework.

Sample pointcut definition	
Component Initialization	`pointcut pcComponentInitialization() : staticinitialization(exp.adapt.component.*);`
Component Invocation	`pointcut pcComponentInvocation() : call(* exp.adapt.component.*.*(..));`
API structures and mechanisms	
Component Identification	`org.aspectj.lang.JoinPoint thisJoinPoint(getThis() and getTarget())`
Argument Values	`org.aspectj.lang.JoinPoint thisJoinPoint.getArguments();`
Method Information	`org.aspectj.lang.JoinPoint.StaticPart org.aspectj.lang.Signature (thisJoinPointStaticPart.getSignature())`
Class Identification and Interface Inspection	`java.lang.reflect.Class java.lang.reflect.Method`

In order to identify component classes and perform interface inspection we will use the Java Reflection API. Through this API we can obtain the class of each component and extract information from it such as name, public attributes, and method signature description. It is worth noticing that parameter name information is not stored in

standard Java .class files, so it is not retrievable using standard Java reflection. However, the AspectJ compiler does enrich compiled classes with that information. We will consider that we have that information readily available for our purposes.

4 Conclusions and open issues

In this paper, we have discussed the potential approaches to Aspect-Oriented Dynamic Component Adaptation in order to support Dynamic Component Evolution, as well as their advantages and drawbacks. We have justified the choice of dynamic adaptor management in a first approach and illustrated its use proposing a design for an adaptation management framework, and how it can be used in order to support the process of component evolution. In order to test this approach we are currently developing a prototype in AspectJ. Although the platform does not support dynamic weaving, it is capable of performing load-time weaving, which is enough in order to test our approach. The ontologies we are planning to use in this prototype will be stored in OWL. This will make it easier to create and read the ontologies since tools and libraries to process OWL are available. So far, only the signature and protocol levels have been tackled, and further study has to be performed related to mapping generation in order to provide suitable techniques for the semantic level as well.

Although our chosen approach suffices the requirements to perform dynamic adaptation, dynamic adaptor generation has a great potential and it is a very promising approach to explore in further work, as compiler and virtual machine technology evolves.

References

1. Allen R. and Garlan D. A formal basis for architectural connection. ACM Trans. on Software Engineering and Methodology, 6(3):213–49, 1997.
2. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. The Journal of Systems and Software. Special Issue on Automated Component-Based Software Engineering 74 (2005), pp. 45-54.
3. Canal, C., Fuentes, L., Pimentel, E., Troya, J.M., Vallecillo, A.: Adding roles to CORBA objects. IEEE Transactions on Software Engineering 29 (2003), pp. 242–260.
4. Canal, C., Murillo, J.M. and Poizat, P. Software Adaptation. in L'objet, 12(1):9-31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities. to appear. 2006.
5. Cazzola, W., Pini, S. and Ancona, M. The Role of Design Information in Software Evolution. In Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05).
6. Filman, Robert E., Friedman, Daniel P.: Aspect-Oriented Programming Is Quantification and Obliviousness. In Mehmet Aksit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004.
7. Magee J., Kramer J., and Giannakopoulou D. Behaviour analysis of software architectures. In Software Architecture, pages 35-49. Kluwer, 1999.
8. McIlraith, S.A., Martin, D.L.: Bringing semantics to Web Services. IEEE Intelligent Systems, 18(1):90-93, Jan/Feb, 2003.
9. "OWL-S: Semantic Markup for Web Services", The OWL Services Coalition (2004), http://www.daml.org/services.
10. Pinto, M.: CAM/DAOP: Component and Aspect Based Model and Platform, PhD thesis. Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga (2004) Available in Spanish.
11. Popovici, A., Frei, A., Alonso, G.: A proactive middleware platform for mobile computing. In: 4th ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil (2003)
12. Yellin, D.M., Strom, R.E.: Protocol specification and component adaptors. ACM Transactions on Programming Languages and Systems 19(2) (1997)