Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 189 (2007) 21–34
www.elsevier.com/locate/entcs

An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution[1]

Javier Cámara[a], Carlos Canal[a], Javier Cubo[a] and Juan Manuel Murillo[b]

[a] *Dept. of Computer Science, University of Málaga, Spain.*
Emails: jcamara@lcc.uma.es, canal@lcc.uma.es, cubo@lcc.uma.es

[b] *Dept. of Computer Science, University of Extremadura, Spain.*
Email: juanmamu@unex.es

Abstract

This paper briefly describes the design of a dynamic adaptation management framework which exploits the concepts provided by Aspect-Oriented Software Development (AOSD), in particular Aspect-Oriented Programming (AOP). The framework uses reflection and adaptation techniques in order to support COTS composition and evolution by tackling issues related to signature and protocol interoperability. This provides a basic infrastructure for a non-intrusive, semi-automatic approach for syntactical and behavioural adaptation.

Keywords: Dynamic Adaptation, Evolution, Framework, CBSD, AOP, Reflection

1 Introduction

One of the most significant trends in the software development area is building systems incorporating pre-existing software components, commonly denominated *commercial-off-the-shelf* (COTS) [18]. 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. However, due to the black-box nature of these components, development teams have no control over their functionality, performance, and evolution. Most of the time these components are not designed to interoperate with each other, requiring customised

[1] This work has been partially supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), project P06-TIC-02250 funded by the Andalusian local Government and project PRI 2PR04B011 funded by the Extremenian local Government.

doi:10.1016/j.entcs.2007.03.026

adaptation which has to be repeatedly performed when teams face component 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. Moreover, some kinds of systems can not be shut down (*i.e.,* banking or airport traffic control systems). Evolving such systems without halting them (*e.g.,* replacing a component with a new version) is a challenging operation which comprises many different problems.

The need to automate the aforementioned adaptation tasks has driven the development of *Software Adaptation* [4], a field characterised by highly dynamic run-time procedures that occur as devices and applications move from network to network, modifying or extending their behaviour. Software Adaptation promotes the use of *software adaptors* [19], which are specific computational entities for solving interoperability problems between software entities (*i.e.,* components, services, etc.). These issues 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 names. This kind of adaptation implies solving syntactical differences such as method names, argument ordering and data conversion and synthesis.
- 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. Mismatch may also occur at this protocol level, because of the ordering of exchanged messages and of blocking conditions. The kind of problem that can be addressed at this level is, for instance, compatibility of behaviour (*i.e.,* whether the components deadlock when combined).
- Semantic Level: This level describes what the component actually does (*i.e.,* its functional specification). Even if two components present perfectly matching signature interfaces and follow compatible protocols, we have to ensure that the components are going to behave as expected.
- Service level: Even if we are able to find a perfect match between components at the signature, protocol and semantic levels, there is still a broad range of mismatch sources, related with non-functional properties like temporal requirements, security, reliability, accuracy, cost, etc. that make composition impossible.

Although signature level is the state of the art in adaptation (*e.g.,* CORBA's IDL-based signature description), several proposals allow the semi-automatic derivation of an adaptor able to solve the protocol mismatch in some circumstances [1,5]. Nevertheless, the resolution o such mismatch implies a previous enhancement of component interfaces with a description of their protocol [3,2,12].

This work is focused in the design of a framework based on Software Adaptation techniques and how these can be applied in order to support *Dynamic 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 must be non-intrusive. In this sense, *Aspect Ori-*

ented Programming (AOP) [15] makes a suitable candidate, providing mechanisms to extend and modify the behaviour of components without directly altering them (*i.e.,* their code). Automatic and dynamic procedures are also required 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 use of such kind of framework can reduce integration effort through the support of (semi)automatic component adaptation in the evolution of non-stoppable COTS-based systems.

In this paper, Section 2 provides an overview of Aspect Oriented Programming. Section 3 briefly describes a small example which will be used to illustrate our proposal throughout the upcoming sections. Section 4 describes the design of a dynamic adaptation management framework based on AOP, automatic protocol adaptor derivation, and illustrates key implementation issues using AspectJ [9]. Section 5 compares our proposal with related work in the fields of Software Adaptation and AOP. Finally, Section 6 presents conclusions and open issues.

2 Overview of Aspect Oriented Programming

Aspect-Oriented Programming (AOP) is based on the idea that systems are better programmed by separately specifying the different concerns (properties or areas of interest) of a system and a description of their relations, and then relying on mechanisms in the underlying environment to *weave* or compose them together into a coherent program. Taking a look at the modular structure of a system, it can be observed that while some concerns are neatly localised within a specific structural module, others cross multiple elements. AOP is focused on mechanisms for simplifying the realization of such *crosscutting concerns* (*e.g.,* security). AOP provides *aspects* as the mechanism to provide an explicit structure for the expression of crosscutting concerns, compacting into a single structure behaviour that otherwise would be scattered throughout (and tangled with) the rest of the code in the system. AOP also provides mechanisms for weaving aspects and base code together into a coherent working system. This weaving process can be performed at different stages of the development, ranging from compile-time to run-time (dynamic weaving)[13]. The dynamic approach implies that the virtual machine or interpreter must be aware of aspects and control the weaving process, although it represents a remarkable advantage, considering that aspects can be applied and removed at run-time. This allows the modification of application behaviour during the execution of the system in a transparent way.

While with conventional programming techniques, programmers have to explicitly call other components' methods in order to access their functionality, the AOP approach represents a remarkable advantage by offering implicit invocation mechanisms for invoking behaviour in code whose writers were unaware of the additional concerns (*Obliviousness*). This implicit invocation is achieved by means of *join points*. These are regions in the dynamic control flow of an application (method calls or executions, exception handling, field setting, etc.) which can be picked up or intercepted by an AOP program by using *pointcuts* (expressions which allow

the quantification of join points) to match on them. Once a join point has been matched, the AOP program can run the code corresponding to the new or injected behaviour (*advices*) typically *before, after, instead* of, or *around* (before and after) the matched join point. Since join points are dynamic, it is possible to expose runtime information such as the caller or callee of a method from a join point to a matching pointcut.

Particularly, component communication can be regarded as a crosscutting concern, whose behaviour can be modified making use of AOP. Hence, components can be wrapped up by aspects able to capture all incoming/outgoing messages by means of pointcuts, and modified conveniently through the application of advice.

3 Running Example

In order to illustrate our approach, we describe an enterprise information system where business rules, rather than being implicit (*i.e.,* not written as rules, but embedded in application logic), are explicit and embedded on a centralised engine for execution. Hence, any business policy can be changed at a single point, and be accessed across the enterprise network. Making business rules explicit in such a way facilitates the use of COTS products, reducing the company's development costs.

This centralised *rules engine* for business rule execution (`RulesEngineComp` component) does not initially incorporate the sets of rules to be executed. Whenever business rules are updated and need to be reloaded, the engine requests a `RuleSetProvider` component which supplies the sets of rules to execute in an appropriate format. This component must be first queried for a rule provider (`getProvider`), which will be used next to create the rule set (`createRuleSet`). Rule sets are ultimately stored in a database which is queried by the `RuleSetProvider` component (`executeQuery`), and then served when finally requested (`getRuleSet`). However, the current `RulesEngineComp` component is limited in performance, so the development team wants to replace it with a more efficient third-party solution. This new rules engine component (`RulesEngine`) is built to directly retrieve rule sets from a database through `loadRules`. The system must be continuously operative, so it cannot be halted in order to perform the substitution of the component. Moreover, as it can be observed in the description of both component interfaces (depicted in Figure 2), the `RulesEngine` and `RuleSetProvider` components are not built to work together, presenting mismatching signatures and protocol.

- Regarding the protocol level, *independent evolution* is given if a message on a particular interface has not an equivalent in the counterpart's interface. Taking a closer look at the component interfaces, it can be observed that `setMode` has no correspondence on the `RuleSetProvider` interface.
- Concerning the signature level, *name mismatch* occurs if a particular component is expecting a particular input message, and receives one with a different name (*e.g.*, `RulesEngine` sends `loadRules` whereas `RuleSetProvider` is expecting `getProvider`). Moreover, it can be acknowledged that the expected function-

ality of `loadRules!` corresponds to several messages on its counterpart interface, and that the required parameters for these messages require type conversion, renaming, and reordering.

Throughout the following sections we introduce our framework used in order to work out the different mismatch situations described above.

4 Dynamic Adaptation Management Framework

Adaptors are automatically built from an abstract description of how mismatch between components or services can be solved (*i.e.,* adaptation *mapping*), which is based on the description of component interfaces. The first step towards the realisation of adaptation is the obtention of this mapping. Anyway, its construction falls out of the scope of this paper. Given a mapping description, this section is focused on the design of an aspect-based adaptation management framework able to work out the mismatch between components at the protocol and signature levels.

4.1 System Architecture

Signature and protocol information from the components being adapted is required to produce a consistent mapping or correspondence between their interfaces. This is obtained from the components using techniques for the incorporation of metadata (specifically annotations) [7]. However, is worth noticing that the available amount information may vary depending on the specific platform where adaptation is being performed, so the need for the incorporation of metadata may vary accordingly.

As it is depicted in Figure 1, the architecture of the system contains three basic functional modules implementing the different concerns comprised by adaptation:

(i) **Interface Manager:** Gathers information about the components' interfaces.

(ii) **Adaptor Manager:** Derives adaptors using the algorithm presented in [1] for the interaction between the components, making use of the aforementioned mappings.

(iii) **Coordination Manager:** Coordinates the interaction between components, translating the messages based on the description of the adaptors previously derived.

The implementation of these tasks, grounded on the principles of AOP, exploit a join point model which enables clean message translation, since components do not need to be internally modified, and pointcut definition provides a compact way to intercept relevant events (component initialization and method invocation are of special interest). Although this framework relies on a standard join point definition language (a thing which usually implies suffering the consequences of structural and syntactical dependency from base code [6,10]), this does not affect the way in which the different managers operate, since the pointcut definitions used are trivial and do not include any specific syntactical nor structural patterns. The implementation of these concerns as aspects, splitting coordination from concerns such as adaptor

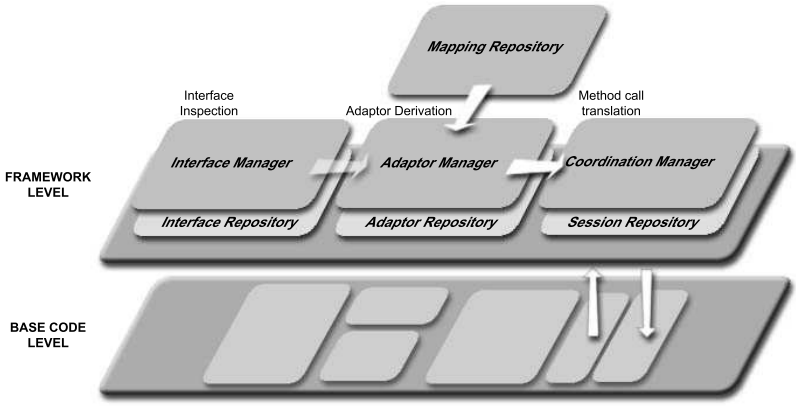

Fig. 1. Framework architecture diagram.

generation, or interface description management grants a clear modularization of the framework.

4.1.1 Interface Manager

It inspects the interfaces of the components as they join the context of the system, and keeps their description in an interface repository in order to use them later for mapping generation. For this purpose we use reflection techniques. Upon initialization of the component c 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.

Since components usually exchange messages in a client-server manner, a complete description of both their offered and required interfaces (*i.e.*, the set of messages received and sent by the component, respectively) is necessary. For instance, in the particular case of Java, the only information available is the description of the messages which belong to the offered interface M_o (through reflection), so the component must be complemented with a description of the signature of its required interface M_r. A complete description of both interfaces must include a minimum set of information for each method consisting on:

- Message (*i.e.,* method) name.
- Ordered parameter names and types.
- Return value types.
- Exceptions raised.

Component interfaces are also extended by including protocol information on their descriptions. The behavioural interface of the components can easily be specified by means of a Labelled Transition System (LTS) [5].

Definition 4.1 [Behavioural Interface] A Component's *Behavioural Interface* is a tuple (M, S, I, F, T) where: $M = M_o \cup M_r$ is an alphabet (set of messages or events), S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times M \times S$ is the transition function.

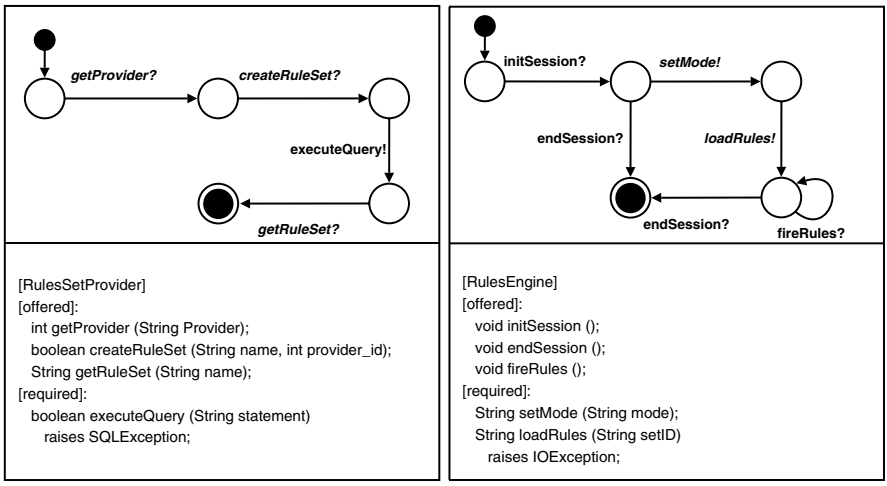

Fig. 2. Component Signature and LTS corresponding to the interface description entries for the `RulesEngine` and `RuleSetProvider` components in the interface repository. Note that emissions and receptions are denoted in the LTS by `!` and `?`, respectively. The messages which correspond to the overlapping parts of the protocol in both interfaces are typed in italic font.

As it can be observed in Figure 2, each of the entries in the interface repository contains a description of both offered and required interfaces and an automaton which specifies the protocol followed by the component.

4.1.2 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 or receive one or several messages to/from other components. Every time one of these messages is generated or received, the manager intercepts it and checks if it is the first one consigned to or received from a target component of class T in the mapping. If that is the case, an adaptor is automatically generated between the source and target component classes making use of the algorithm described in [1]. This adaptor is stored in a repository and it will be used for interaction management between any pair of components of classes (S, T). Once generated, these adaptors allow syntactical adaptation providing message and parameter name translation, data conversion, and parameter reordering. They 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.

A mapping enables outbound messages from a source component to be mapped into different calls to other components within the scope of the system. This mapping contains an initial declaration section, where different values can be defined and modified, such as constants or synthesised values based on different parameters from source messages. Then, the source message is mapped into a sequence of calls to other components in which the actual values for the parameters can either be taken directly from the source message's list or parameters, or from the initial declaration section (synthetic parameters). In Figure 3 we can observe two message bindings which correspond to our example:

```
...
<messageBinding>
   <sourceMessage name="setMode" component="RulesEngine">
      <parameter name="mode" type="String">
      <return type="String"> XML_SERIALIZED </return>
   </sourceMessage>
</messageBinding>

<messageBinding>
   <sourceMessage name="loadRules" component="RulesEngine">
      <parameter name="setID" type="String">
      <exception name="IOException">
      <return type="String"> [call3.return] </return>
   </sourceMessage>

   <call id="call1" name="getProvider" component="RuleSetProvider" returnType="int">
      <callParameter name="Provider" type="String">
         org.jsr94.RuleServiceProviderImpl
      </callParameter>
   </call>

   <call id="call2" name="createRuleSet" component="RuleSetProvider" returnType="int">
      <callParameter name="name" type="String"> [src.setID] </callParameter>
      <callParameter name="provider_id" type="int"> [call1.return] </callParameter>
   </call>

   <call id="call3" name="getRuleSet" component="RuleSetProvider" returnType="int">
      <callParameter name="name" type="String"> [src.setID] </callParameter>
   </call>
</messageBinding>
...
```

Fig. 3. Mapping between the `RulesEngine` and `RuleSetProvider` components. It can be acknowledged how communication direction is reversed *wrt.* the protocol descriptions depicted in Figure 2.

- The first one makes the adaptor accept the `setMode` message issued by the `RulesEngine` component, and return the constant string `XML_SERIALIZED`, which is the expected return value for the invocation, although no effective calls have been performed within the scope of the system. This works out the independent evolution situation described in Section 3.
- The second binding enables the `RulesEngine` component to retrieve a given set of rules when it accepts the `loadRules` message:
  - First, the adaptor calls `getProvider`, using a default provider implementation value (`org.jsr94.RuleServiceProviderImpl`).
  - Next, `createRuleSet` is called, using as parameters the `setID` value supplied by the source message (referenced as `[src.setID]`), and the value returned by the previous call, identifying the provider (which corresponds to `[call1.return]`). Note that all calls are identified by an `id` attribute in order to enable the reference of their returned values from other parts of the mapping.
  - Finally, the adaptor calls `getRuleSet`, and returns its resulting rule set (`[call3.return]`) to the `RulesEngine` component. It can be observed how within the declaration of `sourceMessage` elements, a resulting value can be assigned including a `return` element.

The protocol for the resulting adaptor is depicted in Figure 4. By accessing the Adaptor Manager, engineers can supervise and tune the behaviour of the components by editing the mappings in order to fit specific needs. This capability enables a semi-automatic approach in which the engineer can easily evolve components worrying mostly about coarse-grained issues.

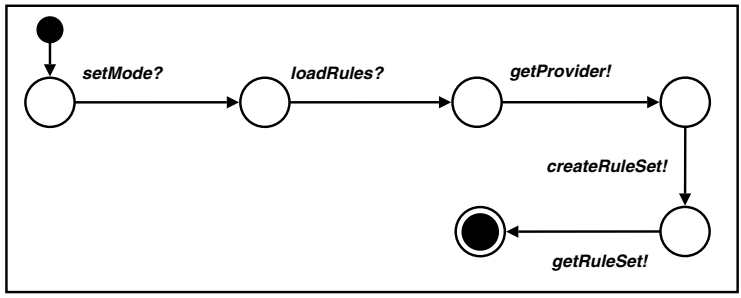

Fig. 4. Adaptor protocol for the mapping represented in Figure 3.

4.1.3 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 adaptor repository. A repository 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 (Figure 5.C). Specifically, each session entry contains a set of information which consists on:

- An instance of all the variables declared within the mapping.
- Actual parameter and return values in component interfaces.
- Current state of the interaction (protocol state in both behavioural interfaces).

This session information is updated if necessary with each message between components. Session information is available to the mechanisms in the coordination manager since some interactions between components might affect others.

4.2 Implementation Issues

In order to illustrate some of the issues related to the implementation of our proposal, AspectJ is used. This is a language level Java AOP extension which is highly representative of the AOP systems currently used. In this section some of the key structures and mechanisms provided to implement the functionality of the adaptation management framework are highlighted.

4.2.1 Extending Component Interfaces

The incorporation of metadata in components has been realised using Java annotations. These are readable at run-time through reflection, hence making all the information required by the framework about components available at run-time. Specifically, we use multi-value type Java annotations, which have multiple data members. We annotate each component by including a custom-defined annotation type (Behavioural Interface Description). Using a run-time `RetentionPolicy` enables the reading of annotations at run-time. As it can be observed in Figure 6, the interface description contains a collection of required methods and a protocol de-

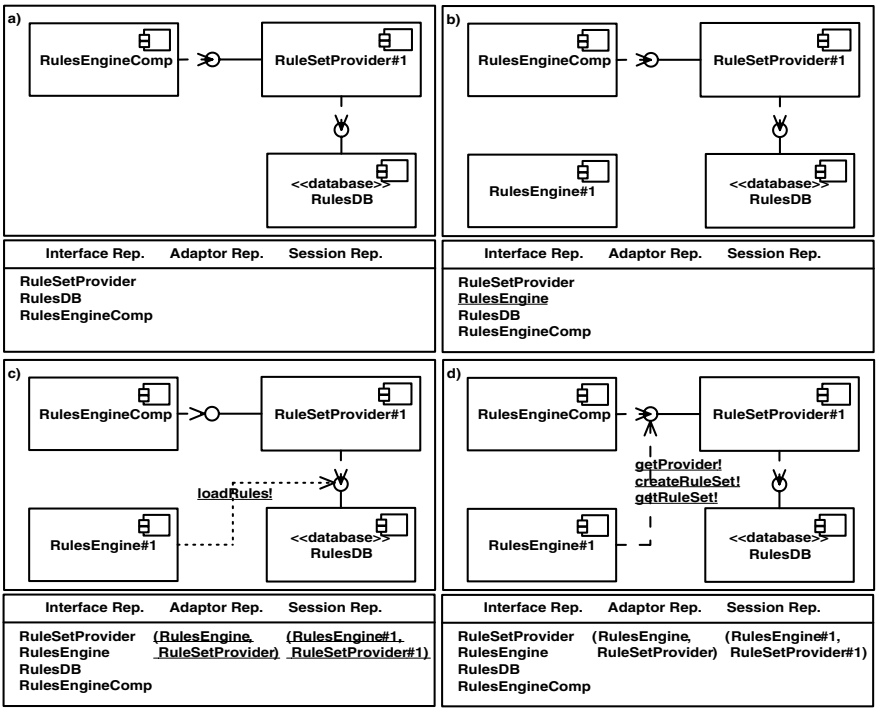

Fig. 5. Simple component interaction example: Initial system context. Interfaces `RulesEngineComp`, `RulesDB`, and `RuleSetProvider#1` are stored in the interface repository **(a)**. Component `RulesEngine#1` joins the context **(b)**. `RulesEngine#1` sends `loadRules!`. Adaptor (`RulesEngine`,`RuleSetProvider`) is generated in the adaptor repository and a session entry for components (`RulesEngine#1`,`RuleSetProvider#1`) is created in the session repository **(c)**.The message is then translated by the coordination manager **(d)**. Note that the figure represents interface dependencies at the different stages of interaction rather than actual message invocations for clarity.

```
@BID ( states={"i:rpA", "rpB", "rpC", "rpD", "f:rpE"},
   transitionf={
         @transition(s1="rpA",m="getProvider",s2="rpB"),
         @transition(s1="rpB",m="createRuleSet",s2="rpC"),
         @transition(s1="rpC",m="executeQuery",s2="rpD"),
         @transition(s1="rpD",m="getRuleSet",s2="rpE")
         },
   required={
         @required (name="executeQuery", parameterNames={"statement"},
                    parameterTypes={"String"},
                    returnType="boolean", exceptions={"SQLException"})
         })
public class RuleSetProvider {
    int getProvider (String provider){...}
    boolean createRuleSet (String name, int provider_id){...}
    String getRuleSet (String name){...}
}
```

Fig. 6. Annotated interface description for the `RuleSetProvider` component. Initial and final states in the protocol are prefixed by `i/f`, respectively.

scription (LTS). Note that the information related to the offered interface is omitted in the BID since it is already available through the standard reflection API.

4.2.2 Framework Implementation

Regarding the framework's design, a minimum set of pointcuts to define in order to provide the required functionality is:

Sample pointcut definition	
Component Initialization	`pointcut pcComponentInitialization() :`
	`initialization(*.*.component.*.new(..));`
Component Invocation	`pointcut pcComponentInvocation() :`
	`call(* *.*.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`

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

- 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 information 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 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 the `getSignature()` method provided by `thisJoinPointStaticPart` is used. Table 1 summarises the main API classes used in the framework.

Component class identification and interface inspection is performed using the Java Reflection API. Through this API the class of each component can be obtained, along with 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.

5 Related Work

The application of AOSD to adaptation is not a new idea [11,17], and currently lots of works on adaptation and dynamic software evolution are based on it.

David and Ledoux present an architecture to manage the adaptation of non-functional concerns [8]. The adaptable concerns are given the shape of an aspect. The proposed architecture supports dynamic adaptation. In [14], Rashid and Kortuem show how aspect oriented techniques can help adaptation in the context of pervasive computing environments. Again the idea is aspectising those facets of the system which could be adapted. In our approach, the main focus is put into using aspects for the implementation of the adaptation framework itself, rather than for aspectising some facets of the system. In our proposal, several precompiled aspects manage adaptation, grouped in different managers which are able to retrieve and interpret the dynamic information required for adaptation.

Redmond and Cahill present in [16] the Iguana/J architecture and programming model to support unanticipated dynamic adaptation. Here, each functional class is associated with a set of adaptation classes which contain the adaptation code. The association is also specified in separated entities in order to achieve improved flexibility. In contrast, in our proposal different adaptors are built and managed specifically for each interaction between components as they join the context of the system. Hence, adaptation code is encapsulated into aspects, although not in a static way. On the contrary, aspects act as interpreters of the design information gathered from the components and as coordinators of the interaction between them.

6 Conclusions and open issues

In this paper, we have discussed our approach to Aspect-Oriented Dynamic Component Adaptation in order to support Dynamic Component Evolution. We have proposed a design for an adaptation management framework, highlighting its advantages as a potential tool to support the process of component evolution. We have then illustrated the foundational differences of our proposal comparing it with related work.

In order to test this approach, a prototype is currently being developed in AspectJ. Although the platform does not support dynamic weaving, it is capable of performing load-time weaving, which is enough in order to prove the operational

basis of the framework. Our main perspective regarding future work, aims at improving this prototype using a middleware which allows dynamic weaving, such as PROSE [13] in its implementation.

Dynamic component adaptation has proved to be a non-trivial problem which requires a vast amount of information about components for them to be successfully adapted in production environments. The most sensible option for the adaptation of COTS products in our opinion is extending their interfaces by including key design information (e.g. protocol, non functional concerns, etc.). Although our current approach suffices the requirements to perform adaptation in simple cases, it is necessary to explore alternatives such as dynamic aspect generation, where adaptors would be implemented by means of aspects which are generated, applied and removed at run-time as required. This approach would increase the complexity of the infrastructure required for execution, demanding some non-trivial modifications to it, such as the inclusion of new functionality (*e.g.,* run-time aspect code generation and compilation). 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 in order to scale up the problem to more complex scenarios (especially in the case of open systems).

References

[1] A. Bracciali, A. B. and C. Canal, *A formal approach to component adaptation*, The Journal of Systems and Software **74** (2005), pp. 45–54, special Issue on Automated Component-Based Software Engineering.

[2] Allen, R. and D. Garlan, *A formal basis for architectural connection*, ACM Transactions on Software Engineering and Methodology **6** (1997), pp. 213–249.

[3] Canal, C., L. Fuentes, E. Pimentel, J. M. Troya and A. Vallecillo, *Adding Roles to CORBA Objects*, IEEE Transactions on Software Engineering **29** (2003), pp. 242–260.

[4] Canal, C., J. M. Murillo and P. Poizat, *Software adaptation*, L'Objet **12** (2006), pp. 9–31, special Issue on Coordination and Adaptation Techniques for Software Entities.

[5] Canal, C., P. Poizat and G. Salaün, *Synchronizing behavioural mismatch in software composition*, in: R. Gorrieri and H. Wehrheim, editors, *FMOODS*, Lecture Notes in Computer Science **4037** (2006), pp. 63–77.

[6] Cazzola, J. J., W. and A. Rashid, *Semantic join point models: Motivations, notions and requirements*, in: *Proceedings of the Software Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT'06)*, 2006.

[7] Cazzola, S. P., W. and M. Ancona, *The role of design information in software evolution*, Proceedings of the Second ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution (RAMSE'05) (2005).

[8] David, P. and T. Ledoux, *Towards a framework for self-adaptive component-based applications*, Distributed Applications and Interoperable Systems. LNCS **2893** (2003), pp. 1–14.

[9] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An Overview of AspectJ*, in: *Proc. of the 15th European Conference on Object-Oriented Programming (ECOOP)*, LNCS **2072** (2001), pp. 327–353.

[10] Koppen, C. and M. Strzer, *Pcdiff: Attacking the fragile pointcut problem*, Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS'04) (2004).

[11] M. Aksit, G. T. and L.Bergmans, *Achieving adaptability through separation and composition of concerns*, Special Issues in Object-Oriented Pogramming (1996), pp. 12–23.

[12] Magee J., K. K. and D. Giannakopoulou, *Behaviour analysis of software architectures*, Software Architecture (1999), pp. 35–49.

[13] Popovici A., A. F. and G. Alonso, *A proactive middleware platform for mobile computing*, In 4th ACM/IFIP/USENIX International Middleware Conference (2003).

[14] Rashid, A. and G. Kortuem, *Adaptation as an aspect in pervasive computing*, Workshop on Building Software for Pervasive Computing at OOPSLA (2004).

[15] R.E., F. and D. Friedman, "Aspect-Oriented Software Development," Addison-Wesley, 2004 .

[16] Redmond, B. and V. Cahill, *Supporting unanticipated dynamic adaptation of application behaviour*, European Conference on Object-Oriented Programming (ECOOP'02). LNCS **2374** (2002), pp. 205–230.

[17] Sánchez, F., J. Hernández, J. Murillo and E. Pedraza, *Runtime adaptability of synchronization policies in concurrent object-oriented languages*, Workshop on Aspect-Oriented Programming at ECOOP'98. (1998).

[18] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," Adisson-Wesley, 2003, 2 edition.

[19] Yellin, D. M. and R. E. Strom, *Protocol specifications and component adaptors*, ACM Transactions on Programming Languages and Systems **2** (1997), pp. 292–333.