

Dynamic Aspects in Large Scale Distributed Applications

An Experience report

Bart Verheecke
Vrije Universiteit Brussel
Pleinlaan 2
1050 Elsene
++32(2)629.38.13

Bart.Verheecke@vub.ac.be

María Agustina Cibrán
Vrije Universiteit Brussel
Pleinlaan 2
1050 Elsene
++32(2)629. 29.64

mcibran@vub.ac.be

ABSTRACT

To realize a high flexibility in service-oriented applications, we propose the Web Services Management Layer (WSML), serving as a middleware layer between client applications and the web services they integrate. To realize the core functionality of the WSML, JAsCo was adopted as a dynamic AOP technology. In this paper, we describe the experiences acquired during the realization of the WSML and its integration with the Service Enabling Platform (SEP) of Alcatel Bell, an open telecom platform for broadband service delivery. In particular, this paper focuses on some of the JAsCo key features that contributed to the realization of the WSML and analyze them in terms of desired software engineering properties. In addition, our experience in developing such a management layer led to various improvements to the JAsCo language and runtime environment, which are discussed in this article as well.

1. INTRODUCTION

This paper describes the experiences acquired on the use of an AOP technology in the context of the research project MOSAIC. This project, carried out in collaboration with Alcatel Bell, focuses on the development of a Web Services Management Layer (WSML), a layer in between the client applications and the web services. The WSML realizes hot-swapping between semantically equivalent services, advanced selection of services based on non-functional properties and client-side management. The WSML is a complex example of the application of dynamic AOP. Its architecture and implementation heavily relies on the use of dynamic aspects. In particular, the JAsCo dynamic AOP language is adopted for the realization of this layer.

The WSML research project serves as a useful, complex and dynamic testing environment for the different features of JAsCo. The applicability of new JAsCo features was tested in the WSML and the need for improvements was motivated as well during its development. The experiences during the realization of the WSML led the incorporation of new features and modifications to the JAsCo language and runtime environment.

In this paper, we present examples of how certain AOP features offered by JAsCo successfully contribute to achieving a solution that put into practice good software engineering properties, such as modularity, expressivity and comprehensibility. In addition, we report on key conflicts among these desired software engineering properties encountered during the development of the WSML. Moreover, we focus as well on the new features that were

envisioned during the realization of the WSML and incorporated in the latest releases of JAsCo.

2. TECHNOLOGIES

The distributed testing environment for the several demonstrators we developed combines a wide range of innovative and state-of-the-art technologies. This section provides an overview of these technologies and introduces JAsCo, the dynamic AOP language used in this setup.

2.1 Web Services Management Layer

Web services and Service Oriented Application Development (SOAD) have the ability to change fundamentally the way distributed applications are being developed. Web services are modular applications that are described, published, localized and invoked over a network. The aim of Web Service technology is to facilitate integration of different business processes regardless of the software and hardware used. By using a wide range of XML-based standards such SOAP, WSDL and UDDI, web service technology allows to overcome platform and language dependency. Therefore, Web services are a promising technology to design and integrate complex inter-enterprise business applications.

In service-oriented applications a high flexibility is desired to be able to quickly adapt and respond to changes in the environment, such as network problems or failure and unavailability of services, without having to stop the application. The Web Services Management Layer (WSML) [9][3] is proposed as a platform, that allows easy integration and client-side management of web services in client applications. Figure 1 shows the role of the WSML in a service-oriented application. To realize the high flexibility required in the WSML and to cleanly modularize all service related code in the WSML, we observe the need for dynamic Aspect-Oriented Programming (AOP). The following objectives are realized in the WSML using dynamic aspects:

- *The ability to select dynamically the services to integrate:* multiple services, and even service compositions, can be integrated in the WSML to deliver the functionality requested by a client application. To decouple the client from specific services, the notion of Service Type is introduced: a generic specification of the required functionality without references to specific web services. Concrete services can be registered to provide the functionality specified in a service type. Requests coming from clients can be easily redirected to any of the registered services. This hot swapping

mechanism relies on redirection aspects, which define the logic of intercepting client application requests and replacing them with concrete web service invocations. As such, they encapsulate all communication details for a specific service or service composition.

- *The consideration of non-functional properties in the selection of services:* Non-functional properties and Quality of Service constraints (QoS) can guide the selection of the most appropriate services. Selection criteria can be based on properties defined in the services descriptions and that can be retrieved and checked at the moment the criteria are applied (e.g. price, distance). Another possibility is that the properties involved in the selection criteria are not anticipated and defined in the services. These properties depend on the behavior of the service at run-time. Examples of such properties are average response time, number of successful invocations, etc. In the WSML, aspects are used to modularize selection policies. As selection criteria are based on business knowledge, they tend to change faster than the core application. Therefore, the selection aspects need to be dynamically pluggable to cope with this volatility.
- *Client-side web service management:* The WSML encapsulates the implementation of different client-side management concerns. Examples of such concerns are caching, billing, accounting, security, transaction, etc. Ideally, these concerns need to be plugged-in and out at run-time according to the application requirements. To decouple and cleanly modularize client-side management code in the WSML, management aspects are used. For instance, all code that deals with client-side billing of a service resides in a billing aspect. Management aspects can also be used to monitor services. For instance, if a client application requires the fastest service, the performance of the involved services needs to be monitored. It is essential to insert and remove this monitoring functionality on demand, as the desired properties can dynamically change.

Dealing with these issues in the WSML allows weakening the link between the client application and the specific web services as all web service related code is taken out from the client application and placed in the WSML. As a result, applications become more robust and adaptable to changes in the environment. For more details on the architecture and implementation of the WSML, we refer the user to [9][3][11]. A fully implemented version of the WSML was realized in the context of the MOSAIC¹ project and is available at [10]. This prototype is implemented in Java and JAsCo.

2.2 JAsCo

JAsCo is introduced in [6] as an aspect-oriented programming language tailored for the component based context. JAsCo builds on top of Java and introduces two additional entities: aspect beans and connectors. An aspect bean is an extended version of a regular Java bean and allows describing crosscutting behavior by

¹ Funded by the IWT (Instituut voor de aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen), Mosaic Project, Flanders (Belgium)

means of a special kind of inner class, called a hook. Aspect beans are specified independently of concrete component types and APIs, making them highly reusable. A connector on the other hand, is used for deploying one or more aspect beans within a concrete component context. In addition, connectors are able to specify explicit precedence and combination strategies in order to manage the cooperation among several aspects that are applicable onto the same join point. In addition, the JAsCo technology provides an extensive run-time infrastructure. Using this infrastructure, aspects remain first-class entities at run-time and dynamic aspect addition and removal becomes possible.

2.3 Case Study

The Service Enabling Platform (SEP) of Alcatel is an operational architecture and open third-party-ready development environment that enables telecom operators to offer converged communication services to end-users. Web service technology is used to realize the collaboration of several internal and third party network services that handle network-user interactions, user connectivity, information delivery and network resource optimization. Several experiments have been done to integrate the WSML with the SEP in order to realize a platform that:

- is *flexible* to adapt to changes in the business environment
- supports *reusability* and *configurability* of service capabilities
- allows *customization* and rapidly fine-tuning of service offerings
- provides *added value* in the provisioning of services

One of the realized SEP-WSML scenarios is the realization of various payment schemas in a Video on Demand system (VoD). Examples include flat fee payment, one phase and two phase pre-payment, micro payments, post-paid and subscription payment based on the customer rating. For each payment schema, a billing aspect was instantiated that intercepts the call to the network provider when demanding a bandwidth upgrade (in order to stream the movie) and executes the actual billing on the rating engine of the service provider. Next to these fixed payment schemas, additional promotional actions, expressed as business rules, were implemented. For instance, variations exist where every 10th call to the rating engine was blocked, resulting in a free bandwidth upgrade every 10th time. By using dynamic aspects, the fast changing business policies can be easily enforced in the distributed environment, even if they were unanticipated. By representing them explicitly as dynamic aspects, they can be easily added, changed or removed.

3. EXPERIENCES

The WSML research project serves as a useful complex and dynamic testing environment for the different releases of JAsCo. The applicability of new features in JAsCo was tested in the WSML and the need for other features was motivated during the development of the WSML. The experiences during the realization of the WSML led to various improvements and modifications to the JAsCo language and runtime environment. The following section presents some of the features offered by JAsCo that we found useful for the realization of the WSML and analyze them with respect to the desired software engineering properties such as modularity, expressivity and comprehensibility.

Analogously, new motivated features that were or are planning to be incorporated in JAsCo are also presented and analyzed against these properties.

3.1 Aspects on aspects

The redirection mechanism in the WSML allows forwarding the functional requests originated in the client applications to concrete semantically equivalent web services. In order to achieve a high degree of flexibility and configurability in the WSML, one of the objectives during its implementation was the decoupling of the different service concerns in different aspects. Examples are the redirection concern and the fallback management concern. Thus, we encapsulated each concern in a different aspect a *RedirectionAspect* including the forwarding of a request to a concrete web service that tackles that functionality and a *FallbackAspect* implementing a concrete strategy for dealing with failure while addressing that particular web service. This decoupling allows dynamically taking the decision of which fallback strategy is more appropriate for each service.

A different instance of the aspects redirection and fallback exists per each concrete web service. This combination of redirection and fallback concerns realizes hot-swapping in the WSML. In order these two aspects to work together in the correct way, the fallback needs to hook when the redirection aspect is executed, more specifically the fallback replaces the redirection. Specifying aspects on other aspects is hardly ever supported in current aspect-oriented approaches (except for EAOP [4]). JAsCo supports the definition of aspects that hook on other aspects: connectors can be instantiated on joinpoints that identified the execution of hooks. Using this feature the fallback mechanism is implemented by hooking on the replace of the redirection aspect as follows:

```
static connector FallbackConnector {
    FallbackAspect.FallbackHook fallback =
        new FallbackAspect.FallbackHook
        (* RedirectionAspect.RedirectionHook.replace());
}
```

Code fragment 1 – Fallback connector: example of a JAsCo aspect that hooks on another aspect

At the moment the fallback is triggered, first it needs to proceed with the original behavior, i.e., the service redirection. In case an exception is raised during the web service invocation, the fallback aspect notifies the WSML so that a new service is tried out. To realize this, a new redirection aspect that invokes a different web service is activated and the whole redirection-fallback process starts again, now attempting to communicate with the new web service. This means that the original request originated in the client application should now be redirected to this different concrete web service through this newly activated redirection aspect. However, in order to achieve this, an invocation to `proceed()` is not adequate since it would continue with the interrupted redirection aspect execution and not trigger the execution of the original request again, as pursued in this case.

A new feature is needed that allows a new invocation of the original method that triggered the aspect chaining. Using this argument as motivation, the JAsCo language was extended with the new method `invokeAgain`, in order to allow the invocation

of the original method that triggered the aspect chain all over again (available from version 0.5.12). AspectWerkz has recently identified the need for such a feature and added the `invokeOriginalMethod` construct, which allows skipping the rest of the advices and calling the original method directly [2].

Having the possibility of hooking aspects on other aspects is useful in certain situations as illustrated with the redirection and fallback concerns. However, one disadvantage when using the `invokeAgain` feature is that it is not always clear to understand its effect, i.e., which method will be the one invoked again. When looking in isolation to the fallback aspect, it is not straightforward to identify the method that originated the aspect chaining. Thus, analyzability of aspects is reduced since knowledge about “where” the aspects are hooking is needed to understand the behavior of the aspect itself.

3.2 “Complement” stateful aspect feature

Stateful aspects proposed by Douence et al. [5] constitute a formal model for aspects that are triggered on protocol history conditions. JAsCo provides stateful aspects expressions based on this model (available from version 0.5), allowing aspects to be triggered based on conditions that depend on the execution history of an application. For more information on stateful aspects in JAsCo, the interested reader is referred to [8].

From our experience in Service-Oriented Software Development, and more generally Component-Based Software Development, we observe that it is often interesting to trigger aspects whenever a sequence of *not* allowed communications is attempted. For instance, suppose a booking web service that exposes two operations in order to login and book a flight: `login(String userName, String password)` and `book(String flightNo, String customerID)` respectively. Imagine that the service provider is interested in checking that users of this web service invoke the operation `book` only after the operation `login` is invoked first. That is, if the operations are invoked in a different order than the allowed one, then an exception should be thrown or a notification should be sent, for instance. This motivates the need for having a language feature that would allow the specification of the triggering of aspects on *the complement of a protocol*. By complement we mean every joinpoint sequence outside the allowed defined protocol. In order to achieve this, JAsCo stateful aspects were extended with the `complement` construct, which allows executing crosscutting behavior whenever a certain invocation that does not follow the allowed protocol is executed (available from version 0.8). The stateful aspect implementing the introduced example is shown in code fragment 2. In this example, the first allowed operation is the `login`. Once the user has logged in, he/she is allowed to invoke the `book` operation. That means that only the sequence `login > book` is allowed. If these operations are invoked in another order, then the complement behavior will be executed, in this case, a notification that a not-allowed communication was attempted.

```
class ProtocolCheckerAspect{
    hook ProtocolCheckerHook{
        ProtocolCheckerHook(login(..arg),book(..arg),
            methodsContext(..arg)) {
            complement[execute(methodsContext)]:
```

```

    firstOperation:execute(login)>secondOperation;
    secondOperation:execute(book);
}

replace complement () {
    notify("Not allowed invocation attempted");
}
}
}

```

Code fragment 2 – ProtocolCheckerAspect: JAsCo stateful aspect that checks that the web service protocol is respected

A connector instantiating the `ProtocolCheckerAspect` is shown in code fragment 3. As parameters, the two pertinent operations are provided as well as a set containing the operations that are considered to calculate the complement, in this case the set containing the two operation `login` and `book`.

```

static connector ProtocolCheckerConnector{
    ProtocolCheckerAspect.ProtocolChecker checker =
        new ProtocolChecker.ProtocolChecker (
            void webService1.login(String, String),
            void webService1.book(String, String), {
            void webService1.login(String, String),
            void webService1.book(String, String)
        });
}

```

Code fragment 3 – ProtocolCheckerConnector: deploying of ProtocolCheckerAspect using login and book operations of Web Service1

The complement feature allows for instance the easy implementation of run-time contract checking logic for web services and security concerns in web services communications.

Without stateful aspects, workarounds need to be implemented to keep track of the execution history of a system. The addition of stateful aspects in JAsCo improves on expressivity since they allow explicitly capturing the desired execution paths upon which to trigger aspects. The complement feature improves expressivity of AOP languages since it is possible to specify explicitly which the desired communications are, and trigger aspects whenever they are not respected. Modularity of aspect behavior is improved since the desired protocols are nicely expressed in a single place in the aspect definition and not spread around in the aspect code, to keep track of the execution path with Boolean variables for instance. Finally, analyzability and understandability of the aspect code are also improved.

3.3 “After throwing” language construct

Normally, a JAsCo after advice is not executed whenever the joinpoint it is specified on throws an exception. While this is the desired behavior in most cases, it might also be required to specify some after advice in case an exception is thrown. For instance, consider a monitoring aspect that calculates the number of successful and failed invocations of a web service method and the time needed to complete the invocation. Typically, the aspect has a before advice starting a counter when the service is invoked, and after advice stopping the counter, calculating the time and registering the invocation as a success or a failure. However, when the service invocation fails due to network conditions or

service failure, the after advice is never executed. A workaround that would realize this behavior consists of the definition of a `replace()` advice with a try/catch block. To count the number of failures, the following advice would be specified:

```

replace() {
    try {
        proceed();
    }
    catch (ServiceInvocationFailureException error){
        service.numberOfFailures++;
        throw error;
    }
}

```

Code fragment 4 – Workaround for counting the number of failures of a WS invocation

The `proceed()` specifies to continue the chain of `replace()` behaviors or to execute the original behavior in case there is no other `replace` advice specified. As a result, the code specified in the catch block is only executed when an exception of type `ServiceInvocationFailureException` is thrown. However, this `replace()` advice is rather counter-intuitive and its behavior is not clear to understand when analyzing it in isolation. To enhance comprehensibility and expressivity of the JAsCo aspects, a richer after advice semantics has been added in JAsCo version 0.8: a new `after throwing` advice is only executed when an exception is thrown. Our example to count the number of service invocation failures can then be easily expressed as follows:

```

after throwing (ServiceInvocationFailureException
    error) {
    service.numberOfFailures++;
}

```

Code fragment 5 – After throwing advice to count the number of failures when invoking web services

Multiple `after throwing` advices for capturing different types of exceptions can be declared. Similar to AspectJ [1], the `after throwing` advice always rethrows the exception.

In the new version of JAsCo, version 0.8, after advices are executed even if the joinpoint it is specified upon, throws an exception. To specify an advice that only executes when the joinpoint does not throw an exception, a new construct `after returning` has been added, as the complement of `after throwing`. The new version of AspectWerkz [2], version 2.0, recently introduced a similar feature.

Note that it can also be desirable to specify an `after throwing` advice that does not rethrow the exception. Consider the example of the fallback aspect discussed in section 3.1. In this case, the exception thrown by the redirection aspect (caused by a failure of a service invocation), should be captured and not rethrown by a fallback aspect. The next version of JAsCo, version 0.9, envisions the introduction of such a feature. Although it is very expressive to have different language constructs to distinguish between all these different semantics, it might affect the understandability of the AOP language, since the programmer needs to understand the difference between many similar features to decide which one is the most appropriate in each situation.

3.4 Advanced event handling for aspects

In JAsCo, the reusable crosscutting concerns are written down in aspects, while the runtime deployment details of the aspect are put in connectors. These connectors can be easily enabled and disabled to stop temporarily the execution of the advices specified in the aspect. However, several scenarios were encountered where enabling, disabling and even removing an aspect required some additional behavior to be executed. For example, the monitoring aspect from the previous section, adds new properties, which represent the runtime behavior of the web services (e.g. number of failures, response time, etc). These properties are added to the appropriate `WebService` object. However, when the monitoring aspect is removed, these properties need to be removed too. In addition, when the aspect is temporarily disabled, the properties need to reflect this status. For this purpose, the JAsCo runtime environment now throws an event whenever an aspect is added, removed, enabled or disabled. Inside the aspect, additional behavior can be specified that needs to be executed in these cases. Inside the service monitoring aspect, the following code has been added:

```
public void globalPropertyChangeEvent
    (PropertyChangeEvent event) {

    if(event == HookPropertyChangeEvent.IS_ENABLED)
        service.enableMonitoredProperties();
    else if
        (event == HookPropertyChangeEvent.IS_DISABLED)
        service.disableMonitoredProperties();
    else if
        (event == HookPropertyChangeEvent.IS_REMOVED)
        service.removeMonitoredProperties();
}
```

Code fragment 6 – Advanced Event handling for JAsCo aspects

This simple example illustrates the need for more advanced control over the enabling, disabling and removal of aspects. A workaround for this feature would consist of the specification of an aspect that specifies advices that hook at the moment the aspect is enabled, disabled or removed in the JAsCo runtime environment. Clearly, this solution would be awkward since the aspect would need to hook on a joinpoint outside the core application. On the contrary, the code in fragment 6 is much cleaner and easier to understand. As such, this feature has contributed to the expressivity of the JAsCo aspects.

3.5 Feature Interaction Solutions

When multiple advices are specified to be executed on a single joinpoint, their order of execution can be explicitly specified in a single JAsCo connector:

```
Hook1.before();
Hook2.before();
Hook2.replace();
Hook1.replace();
```

Code fragment 7 – Specifying explicit order in a JAsCo connector

This allows having a tight control over the execution of the aspect behavior. It also provides a partial solution for the *feature interaction problem* [7]. While this approach proves to be sufficient in most cases, it also has the drawback that removing or adding an aspect involves rewriting and recompiling the connector. In some situations, this expressive way of specifying

aspect execution order is not required: a simple ordering of the connectors in the internal JAsCo connector registry is enough. Furthermore, in some cases it is more important to instantiate aspects in separate connectors in order to allow a fast addition, removal, enabling and disabling of individual aspect instances. To facilitate the management of these connectors, *connector priorities* and *connector combination strategies* were added.

Connector priorities allow controlling the execution order of advices that are instantiated in separate connectors and defined on the same joinpoint. The higher the value, the higher the priority. Through the JAsCo connector registry API, the priority of connectors can be dynamically altered, resulting in a dynamic reordering of the connectors. This feature is particularly useful to implement service selection guidelines [11]. A guideline specifies the preference of one service over another one based on some non-functional property, e.g. prefer the use of the cheapest service when possible. This can be achieved straightforwardly by assigning specific priorities to the connectors of the redirection aspects that reflect the order implied by the guideline. More advanced control is introduced by connector combination strategies. Likewise to normal combination strategies, connector combination strategies can be implemented in plain Java and allow filtering the list of all connectors at each encountered joinpoint.

These features are included in the JAsCo language in order to improve modularity of crosscutting concerns and evolvability of the overall solution. They allow the addition and removal of aspects independently of each other in a much easier way while it is still possible to express aspect interactions in a modular way.

3.6 Jasco runtime improvements

By default, the JAsCo connector registry looks for new connectors with specified intervals, allowing for easy loading and removal of aspects at runtime. However, in the context of the WSML, a tighter control on the addition and removal of connectors is needed. The WSML is an aspect-oriented layer on top of the JAsCo runtime environment that controls the compilation and instantiation of aspects and the automatic generation and compilation of the connectors. Therefore, we decided to disable the hot-deployment of connectors and replace it by a mechanism that delegates the control of loading and unloading connectors to the WSML by means of the use of the Connector Registry API. This mechanism also had the advantage that it avoids the possibility of loading illegitimate connectors and aspects into the system. Before, it was possible to load malicious aspects that were not fully tested and that could crash the whole system. With the new mechanism, only aspects that were thoroughly tested and registered in the WSML could be instantiated.

An aspect causing a crash of the system had far-reaching consequences. Not only did this result in downtime of the server, but it also took a long time to reboot the system and to recompile and reload all aspects and connectors. In a small scenario with 10 connectors instantiating 7 aspects, the loading time of the WSML was more than 3 minutes on a Pentium 4 with 1GB of RAM. Therefore, a caching mechanism is provided as part of the WSML in order to reuse already compiled connector and aspect classes. This reduced the loading time by a factor of 10. To avoid aspects that could crash the whole system, all aspects were implemented

following a coding convention. This allows detecting the aspects that cause exceptions and immediately removing them from the runtime environment. The code fragment 8 illustrates this with a replace advice:

```
replace() {  
    try {  
        //do something  
    }  
    catch (Exception e) {  
        WSML.unloadConnector(name);  
    }  
}
```

Code fragment 8 – Coding convention for aspect advices

The incorporation of these improvements to the JAsCo runtime environment contributed to achieving a more robust implementation of the WSML and enhancing evolvability of the overall solution.

4. CONCLUSIONS

In this paper, we presented our experiences gained during a project where we incorporated the WSML, a platform for web services management, and the SEP, an open architecture for telecom operators to deliver added value services to end users. The integrated platform offers flexible solutions in a multi-service context where it could play a central role in the access and edge networks of the future. In this paper, we discussed the use of the dynamic AOP language JAsCo in this environment in terms of desirable software engineering properties such as expressivity, evolvability and comprehensibility. It is clear that a complex distributed application such as the SEP-WSML is a good case study to learn about the advantages of dynamic AOP in a real-world application. It also shows that interaction towards the AOP community is necessary in order to incorporate new desirable language and runtime environment features. In this particular case, bidirectional communication proved to be useful for both the developers of JAsCo as the SEP-WSML was a very interesting testing platform, as well as for Alcatel Bell as this was a good opportunity to learn about the advantages of AOSD for service delivery. A new project will start in 2005, focusing on the use of dynamic AOP for service creation.

5. REFERENCES

- [1] AspectJ, <http://eclipse.org/aspectj/>
- [2] AspectWerkz, <http://aspectwerkz.codehaus.org/>
- [3] Cibrán M. A, Verheecke B. and Jonckers, V., “Modularizing Client-Side Web Service Management Aspects”, in Proceedings of the 2nd Nordic Conference on Web Services, Växjö, Sweden, November 2003.
- [4] Douence, R., Fradet, P. and Südholt, M. A, “Framework for the detection and resolution of aspect interactions”, in Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE), October 2002.
- [5] Douence, R., Fradet, P. and Südholt, M., “Composition, Reuse and Interaction Analysis of Stateful Aspects”, in Proceedings of the 3th International Conference on Aspect-Oriented Software Development”, Lancaster, UK, March 2004.
- [6] Suvéé, D. and Vanderperren, W., “JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development,” in Proceedings of Second International Conference on Aspect-Oriented Software Development, Boston, USA, March 2003.
- [7] Pulvermüller, E., Speck, A., Coplien, J.O., D'Hondt, M. and De Meuter, W., “Proceedings of the Workshop on Feature Interaction In Composed Systems”, European Conference on Object-Oriented Programming, Budapest, Hungary, 2001.
- [8] Vanderperren W., Suvee, D., Cibrán M. A. and De Fraine B., “Stateful Aspects in JAsCo”, submitted to the Workshop on Software Composition, ETAPS 2005, April 2005 Edinburgh, Scotland.
- [9] Verheecke, B. and Cibrán, M. A., “AOP for Dynamic Configuration and Management of Web services in Client-Applications”, Published in the International Journal on Web Services Research (JWSR): Volume 1, Issue 3, July-Sept 2004
- [10] Verheecke, B. and Cibrán, M. A., “Web Services Management Layer (WSML)”, <http://ssel.vub.ac.be/wsml/>
- [11] Verheecke, B., Cibrán M. A. and Jonckers V., “Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection”, Proceedings of the European Conference on Web Services 2004 (ECOWS'04), Erfurt, Germany, September 2004.