Next Generation Aspect Oriented Middleware Workshop

NAOMI 2008

AOSD Conference 2008 Brussels

Seventh International Conference on Aspect-Oriented Software Development

1st April 2008
Workshop Organisation

Workshop Chair

- Wouter Joosen (KULeuven, Belgium)

Program Committee

- Jörg Kienzle (McGill University, Canada)
- Renaud Pawlak (Rensselaer at Hartford, USA)
- Gordon Blair (Lancaster University, UK)
- Frank Eliassen (University of Oslo, Norway)
- Eddy Truyen (KULeuven, Belgium)
- Bart De Win (KULeuven, Belgium)
- Johan Fabry (University of Chile, Chile)
- Monica Pinto (Malaga University, Spain)
- Lydia Fuentes (Malaga University, Spain)
- Paul Grace (Lancaster University, UK)
- Michael Haupt (Hasso-Plattner-Institut, Germany)
- Renato Cerqueira (PUC-Rio, Brazil)
- Shigeru Chiba (Tokyo Institute of Technology, Japan)
- David H. Lorenz (The Open University of Israel)
- Gunter Kniesel (University of Bonn, Germany)
- Adrian Colyer (Interface21)
- Yvonne Coady (University of Victoria, Canada)

Organisation Committee

- Geoff Coulson (Lancaster University, UK)
- Bert Lagaisse (KULeuven, Belgium)
- Phil Greenwood (Lancaster University, UK)
- Frans Sanen (KULeuven, Belgium)
- Bholanath Surajbali (Lancaster University, UK)
Introduction

Practical middleware platforms nowadays are highly complex. This complexity surfaces internally in middleware construction, and externally in the programming models supported and features/services offered. There is a need to reduce this complexity both internally and externally: internally through enhanced modularity, and externally through the use of contemporary composition paradigms. Furthermore, middleware platforms need to be configurable, reconfigurable, customisable and extensible to reflect the specific needs of applications, the heterogeneity of involved systems and the multiple distributed deployment contexts.

Aspect-Oriented Software Development (AOSD) has been put forward as a promising and successful paradigm to improve the reusability, extensibility, modularity and customization capabilities of middleware platforms. Moreover, most non-trivial applications of AO have emerged from the middleware domain, also a number of AO-frameworks (e.g. JBoss and Spring) specifically target the AO-middleware application domain.

The goal of this workshop is to bring together and extend the AOSD and middleware research communities. This hybrid AO-middleware community is a cross-fertilization of 1) the AOSD community which has a plethora of promising technologies for further research on middleware development and middleware application composition and 2) the middleware community which has challenging applications for the validation and enhancement of AO-technologies.
## Table of Contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Authors</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design and Development of a Context Oriented Language for Middleware Based Applications</td>
<td>Andrea Sindico, Giovanni Bartolomeo, Vincenzo Grassi, Stefano Salsano</td>
<td>1</td>
</tr>
<tr>
<td>A Light-weight Load-time Weaving Approach for OSGi</td>
<td>Thorsten Keuler, Yury Kornev</td>
<td>6</td>
</tr>
<tr>
<td>A Feature Model of an Aspect-Oriented Middleware Family for Pervasive Systems</td>
<td>Lidia Fuentes, Nadia Gamez</td>
<td>11</td>
</tr>
<tr>
<td>Building a Distributed AOP Middleware for Large-Scale Systems</td>
<td>Ruben Mondejar, Pedro Garcia, Carles Pairot, Antonio F. Gomez Skarmeta</td>
<td>17</td>
</tr>
</tbody>
</table>
Abstract

Nowadays context-aware adaptation is becoming an important feature for pervasive computing applications. In this paper we present JCOOL, a COntext Oriented Language tailored to handle context awareness in Java applications. JCOOL exploits Aspect Oriented techniques so that context changes detection and related adaptations can be considered as two separated crosscutting concerns with respect to the core “business logic” of new or legacy Java applications. Moreover, mobile and pervasive applications generally rely on middlewares that hide the complexity of the underlying environment. In order to show how JCOOL support can be introduced into middleware based application, in the second part of the paper we also describe JCOOL integration in SMILE [1], a Middleware Independent Layer developed in the scope of the SMS project [2].

Categories and Subject Descriptors: D.3.2, D.3.3 [Language Classifications, Language Constructs and Features]: Specialized application languages – Frameworks.

General Terms: Design, Languages.

Keywords: context awareness, aspect oriented programming, domain specific language, middleware.

1. Introduction

Specific mechanisms and API are needed to support context dependent modifications of the behavior of mobile and distributed applications. Existing platforms that try to achieve this goal using general-purpose languages (GPLs), suffer from the common difficulties of GPLs related to the lack of semantic expressiveness of their constructs. Besides, the adaptation to different contexts can be considered as an orthogonal task with respect to the core application logic [3]. In this respect, Object Oriented GPLs suffer from their inability to encapsulate crosscutting concerns, such context awareness, without affecting the components business logic. This suggests the adoption of a Context Oriented Programming approach based on the use of Domain Specific Languages (DSLs) tailored for the context awareness needs; these languages can better capture the crosscutting nature of context awareness and provide more effective constructs to aid the developer in tackling this concern.

This paper describes an ongoing work on the definition of a context oriented language named JCOOL (Java COntext Oriented Language) we have recently started to design and develop as a follow up of the work made in [4]. One of the main goals of JCOOL is the possibility of introducing context awareness capabilities into an already existing Java application without changing its original code. To show how this can be achieved we propose an example of JCOOL integration into SMILE [5][6], a “Simple Middleware Independent LayEr” between applications and the underlying middleware platform. The goal of SMILE is to relieve the developer from the need of writing middleware specific code, focusing instead on the implementation of the application business logic.

2. Related Works

Context-oriented Programming (COP) is a new programming approach which aims to alleviate the spreading of context-dependent behaviours throughout a program by incorporating context as a first-class construct of a programming language [7][8][9]. In [10][11] the following list of mechanisms a Context Oriented Programming Language should provide is described:

- Behavioral variations: variations typically consist of new or modified behaviour of the system components;
- Layers: Layers group related context-dependent behavioural variations;
- Activation: Layers aggregating context-dependent behavioural variations can be activated and deactivated dynamically at runtime. Code can decide to enable or disable layers of aggregate behavioural variations based on the current context;
Aspect Oriented Programming (AOP) [12] can be exploited to address these requirements. For example, Layers of behavioural variations can be realized by the definition of ad hoc around advices whose activation is triggered by other Aspects which play the role of Context monitors as explained in [4]. However, AOP languages only consider the elementary events in the execution flow of a program such as method calls, field accesses, and so on, which in AOP terminology are called join points. AOP join points are not expressive enough to cover the complexity of a context definition which instead may depend on complex and distributed properties of the system components and even of its execution environment. In [3], E. Tanter et al. point out that AOP languages are also limited with respect to the kind of context dependencies that can be expressed. For example, even though there are a number of AOP languages that make it possible to define pointcuts which depend on past execution history, because of the lack of an explicit context definition, they only consider simple events such as method invocations but don’t consider past contexts.

Tanter et al. also propose a list of characteristics a context definition should have:

- **Stateful**: a context may have state associated with it;
- **Composable**: different context definitions can be combined to define complex contexts;
- **Parametrized**: context can be defined generically, and parametrized by aspects that are restricted to it;
- **Scoping**: The scope where layers are activated or deactivated can be controlled explicitly.

In [3], a Reflex extension is proposed which addresses the above requirements. Reflex itself is a Java extension which provides building blocks for facilitating the implementation of different aspect oriented languages so that it is easier to experiment with new AOP concepts and languages. In the framework described in [3] the developer has to define how and when a context has to be saved so that it will be possible to refer to it in a future instant. In this respect, JCOOL makes it easier to refer to past contexts. For example the developer does not have to define how context must be saved because the only informations a context should provide, except his state, are the actual parameters which has verified it. How these parameters are stored is hidden to the developer by the JCOOL underlying environment.

In [13] Costanza et al. describe ContextL, a Context Oriented Programming Language for Common Lisp Object System, which provides a set of language constructs that allow the developer to associate partial class and method definitions with layers. Layers can then be activated and deactivated in the control flow of a running program. When a layer is activated, the partial definitions become part of the program until this layer is deactivated.

The main difference between ContextL and JCOOL is that ContextL does not provide language constructs to define a Context and its inner states. ContextL only provides a macro, named with-active-layers, to activate a layer in the dynamic scope of a program. However, even though the developer does not have to spread context adaptation code in the base program, which is instead encapsulated in the layer definitions, it has to spread with-active-layers block of code in the base program in those points where the related context changes. JCOOL provides distinct language constructs for Context Monitoring and Context Adaptation. It considers these two concerns as two crosscutting concerns: the former crosscuts the base program to detect when it is in a context of interest, the latter crosscuts the Context Monitors to introduce context adaptations when needed. In this way we achieve a strong separation between when and how context adaptation should be carried out. Moreover, these two concerns are well encapsulated in two distinct first-class language constructs. Thanks to this, the base program is not affected by any of these two concerns. Moreover, because of the lack of an explicit context definition, ContextL does not address the Context definition requirements proposed in [3], which are instead explicitly taken into account by JCOOL.

3. **JCOOL**

JCOOL is a domain specific aspect oriented language derived from the UML Profile for context awareness described in [4]. In JCOOL there are two main constructs named Context and Adaptor that are the code level counterparts of the ContextMonitor and ContextAdaptor elements defined in the aforementioned UML profile. As its UML equivalent, a Context is composed by a set of rules which specify conditions that must hold to introduce some kind of context adaptation. In JCOOL each Context is identified by a unique name and can involve one or more components of the base system. This means that a Context definition can affect only those classes of the base system that are listed after the key word involve. A Context is represented as a state machine with a default start state and one ore more states in which it may migrate. To this end, a state transition rule is associated to each state. The relation between a state and its transition rule is expressed with an Horn Clause in a Prolog-like syntax [14].

\[
\text{stateName} :- \text{stateTransitionRule}
\]

A Context is in a given state until the related transition rule holds while it is in the default state if none of its transition rules is verified. A state transition rule consists of a set of one or more predicates, over the components involved by the Context, combined with the logical operators “,”(AND), “|” (OR) and “!” (NOT). Because of these characteristics JCOOL’s Contexts can be considered Stateful thus addressing the first requirement defined in [3].

```
Context SimpleContext involve ClassA {
    default;  
    stateA :-ClassA .attribute == 1 ;
    stateB(i) :-ClassA .attribute == 2 ;
}

Context ComplexContext involve ClassB, SimpleContext
{
    default;  
    compositeState:-SimpleContext.stateB8,
                  (ClassB .attributeB >1);
    complexState:-((SimpleContext.stateA)+,
                   (SimpleContext.stateB)\{2\}),
                   (SimpleContext.stateA);
}
```

Figure 1. Examples of Contexts
Contexts are also Composable, because they may be built as a composition of other contexts. The states of a composite Context have state transition rules that depend on the state transitions of the Contexts it is composed by.

Figure 1 depicts two examples of Context definition. The first one consists of a Context named SimpleContext which involves the ClassA class of an hypothetical base system. This context can be in two different states: stateA and stateB, depending on the value of an attribute of a ClassA's instance.

The second Context depicted in Figure 1, named ComplexContext, is an example of composite context because it depends on the SimpleContext context and on the ClassB class. It starts in the default state but migrates in the composite state as soon as the SimpleContext is in the stateA state and the value of an attribute of a ClassB's instance is greater than a certain value.

Sometimes it could be necessary to detect a precise sequence of events in order to consider a Context in a certain state. To this end, square brackets must be used to enclose those events of a state transition rule that must occur in the exact sequence they are written in. Operators ?, + and * can be used, like in regular expression, to express that an event should occur respectively: never or one time; at least one time; never or any time. Curly brackets can be used to enclose the exact number of times an event must occur. In composite context this syntax can be used to define a context state which depends on an exact sequence of past contexts and possibly refers to their context parameters. For example, the context ComplexContext of Figure 1 migrates in the complexState only after that the SimpleContext has migrated into the stateA at least one time, then it has migrated in the stateB two times and it is currently in the stateA.

On the transition between two states, a context may trigger the execution of one or more Adaptors through the invocation of one of its entry points (Figure 2). As its UML counterpart, an Adaptor is a container for context adaptation mechanisms. Each Adaptor is identified by a unique name and may be driven by one or more Contexts, as well each Context may drive several Adaptors. Parameters can be passed to the adaptation action after a transition rule is evaluated and fired. These parameters can be free variables which take the values of those objects which verify the fired state transition rule. For example, the i variable used in the SimpleContext.stateB definition, takes the value of the ClassA's instance which verifies the related state transition rule when fired.

An Adaptor has as many entry points as the state transitions it is designed to intercept. For each entry point two kinds of adaptation can be defined: one shot activities and behavioural variations. One shot activities consist of two pieces of code associated to an Adaptor's entry point: the former must be executed at the related context-state incoming event (in); the latter has to be executed at the related context-state outgoing event (out). Within these blocks it is possible to use the optional parameters passed with the state transition.

Behavioral variations, or layers, consist of a set of alternative method definitions that may affect classes or particular class instances passed as parameters to the Adaptor. A behavioural variation is active until the involved Context remains in the related state. When a behavioural variation is no longer active the methods it has affected return to their original implementations.

The difference between these two kinds of adaptation is that one shot activities are executed as soon as the related Context goes in/out a certain state. They can use objects passed by the related context to perform activities preparatory to the context change. Behavioural changes, instead, have to be considered as a dynamic override of some methods of objects or classes of the base system that change their behaviour until they remain in a certain context state. As mentioned, behavioural changes may affect classes or instances so that, in a given time, different objects of the same class may have different implementations of the same methods depending on their context. When a behavioural variation is removed the methods it has affected return to their original implementations.

Figure 2 depicts an example of Adaptor which is driven by the SimpleContext of Figure 1. As soon as the SimpleContext enters in the stateA the in block of code of the related SimpleAdaptor entry point is executed. Until the SimpleContext is in that state, behavioural changes are introduced that consist, for this example, in the overriding of the ClassA.simpleMethod method. Since this behavioural variations is related to the ClassA it affects all the ClassA instances of the system. When the SimpleContext goes out the stateA the out block of code of the related SimpleAdaptor entry point is executed and the behavioural variations are deactivated so that the ClassA.simpleMethod returns to its original implementation.

4. SMILE

Developed in the scope of the SMS Project [2] SMILE is an abstract platform [15] with the explicit goal of avoiding developers to rewrite their applications as a consequence of changes in the underlying middleware, allowing to focus the development effort on the business logic more than on the implementation details. An application written for SMILE consists of a set of peers, named SMILEPeers, which are abstract classes loose coupled with the underlying runtime environment.
From the developer point of view, SMILE peers are autonomous entities which may communicate through message exchanges. Each peer may access a common minimum set of features provided in form of an API. These features typically include naming and addressing, service registry, message routing mechanisms, etc. and are implemented by exploiting the underlying middleware facilities.

In order to exploit these facilities, without directly relying on them, SMILE provides a mechanisms called binding, similar to the one defined in Web Service Description Language (WSDL) [16]. Thanks to this separation layer, applications written for SMILE are not to be changed as a consequence of changes in the underlying middleware platform; instead only the binding has to change. Unlike WSDL, however, the same SMILE application, at run-time, might exploit more than one binding, thus dynamically adapting its behaviour to different contexts. More details can be found in [5][6].

In the following section we will describe how this feature represents an interesting use case for JCOOL, which may be seamlessly integrated into SMILE. For clearness’ sake, with the help of an example, we will describe step by step the procedure developers have to follow to successfully achieve such an integration, together with some internal details the platform hides them, in order to properly run JCOOL context oriented applications in a seamless way.

5. JCOOL as COP Support for SMILE

Developers wishing to use JCOOL support in SMILE first have to identify possible join points; in addition to application specific operations, these include specific pointcuts provided by the SMILE API, which are of four kinds: callbacks for implementing the application lifecycle; methods to interact with the service registry; methods and callbacks for message exchanges and for remote procedure calls; interfaces between the applications and the bindings. Subsequently, developers add two additional sets of files to their SMILE application source code: one set defining Context, with initial state and state transition rules; the second set defining the Adaptors. Both these file sets have a global scope, i.e. they can refer to any object (including custom objects) defined in the sources.

As an example, consider a SMILE application composed by a set of SMILE Peers with some of them having to send messages requiring a high privacy level. JCOOL can be used to introduce a context aware adaptation so that, depending on the reliability of the transport protocol available in the currently active binding, a SMILE Peer should send or not its message.

Figure 3 depicts the definition of a JCOOL Context named MediumReliability which involves the SMILEPeer class. Suppose this context can be in two different states: low and high, depending on the security level provided by the transport protocol used by a given binding.

The instance parameter, used in the context definition, is a formal parameter which is evaluated whenever the state transition rule is fired. Once evaluated, it is passed as actual parameter to any Adaptor triggered by this Context. In this example, whenever the MediumReliability context migrates into the low state, it triggers the execution of PrivacyAdaptor. PrivacyAdaptor prints out a message to alert about the context change and introduce a behavioral variation that changes the send method of the passed SMILEPeer instance so that this instance will not send any message requiring a high privacy level. Note that this change affects only SMILEPeer instances which are using an unsecure binding whereas other SMILEPeers continue to use seamlessly their original implementation of the send method.

The SMILE platform takes care of implementing such logic seamlessly, in two simple steps. At compile time, JCOOL Adaptors pass through an ad hoc pre-processor that weaves them with legacy sources in order to insert adaptation code. At runtime, an entity called “Broker”, implementing inversion of control and listening at any event related to peers contained in a given platform instance, is responsible also to monitor the bindings to the underlying middleware platforms. Whenever the application uses JCOOL support, context states and transition rules contained in a JCOOL Context are dynamically interpreted by the Broker which finally invokes the execution of triggered adaptation actions whenever needed.

```
Context MediumReliability involve SMILEPeer{
  default;
  low(instance)::=instance.currentBinding.
    equals(SipBinding);
  high(instance)::=instance.currentBinding.
    equals(XMPPBinding) &&
      (getTransport().
        equals(XMPPBinding.HTTPS));
}

Adaptor PrivacyAdaptor {
  MediumReliability.low(instance) {
    in : { // No action }
    // Layer definition
    public void instance.send(Message msg){
      if(msg.getOverallPrivacyLevel()==HIGH){
        msg.getSender().printMsg("Message privacy level not compliant with the current medium");
      } else { proceed(); }
    }
  }
}
```

Figure 3. Examples of JCOOL in SMILE

6. Conclusions

The ultimate goal of research in Context Oriented Programming is to provide language constructs to aid software developers in a better encapsulation of crosscutting context dependent behaviors. In this paper we have presented JCOOL, a domain specific language that makes possible a strong separation between the Context Monitoring and Context Adaptation concerns with respect to the base system. This feature aids the designer to think at these two concerns separately, designing different Context Monitors and Adaptors that can even be reused and combined into different architectures to achieve the desired degree of context awareness. Moreover, in order to show how JCOOL support can be provided into a middleware for distributed applications, we have also described JCOOL integration into SMILE.
What we have presented is a first step of an ongoing work. In the future, we intend to investigate about the possibility to import Prolog knowledge bases in a Context definition so that its state transition rules, thank to their horn clause syntax, may use *modus ponens* to detect inferred context states. For example: if the fact “Paris is in France” is known, the context *location*("Paris") will be accepted as a match of the context *location*("France") [9].

Another open issue concerns the coincidental activation of different behavioural variations that affect common target components; i.e. different behavioural variations that affect the same methods of the same components at the same time. Currently, for each component, behavioural variations affecting the same method are activated in a stack like way so that when a behavioural variation is activated it automatically deactivates the previous one. However we would investigate about a better way to solve this issue i.e. by providing a way to automatically merge in a unique variation independent not conflictual variations that affect the same components.

We are currently working on the development of a first prototype of JCOOL pre-processor that will perform a static weaving of Context and Adaptors’ code with a given target base system. This first goal will not cover the possibility to handle unforeseen context adaptation. However, to address the issue of unpredictable context changes and related adaptations, we are already investigating about the possibility to exploit runtime weaving capabilities of modern AOP environment [17].

References


A Light-weight Load-time Weaving Approach for OSGi

Thorsten Keuler
Fraunhofer Institute for Experimental Software Engineering
Fraunhofer-Platz 1
67663 Kaiserslautern
(+49)631 6800 2162
thorsten.keuler@iese.fraunhofer.de

Yury Kornev
Fraunhofer Institute for Experimental Software Engineering
Fraunhofer-Platz 1
67663 Kaiserslautern
yury.kornev@iese.fraunhofer.de

ABSTRACT
In the context of dynamic, component-based systems, the OSGi platform (Open Standard Gateway Initiative) supports the dynamic deployment of software services in form of so-called bundles. Since aspect-oriented programming offers facilities for managing crosscutting code adaptations efficiently, OSGi needs to be equipped with that kind of capabilities. Existing approaches (e.g. JBOSS-AOP) for aspect-oriented programming, however, have to be adapted due to restricted class visibility within OSGi. To that end, approaches that are intended to be used with OSGi have to be supported in terms of physically separating aspect code from concrete pointcut implementations. Existing efforts that aim at integrating AOP technology with OSGi have shown some insufficiencies. Some even require the OSGi specification to be extended. In this paper, we show a concept that enables a light-weight integration of AOP by enabling customized load-time weaving within OSGi without having to change the platform at all.

Keywords

1. INTRODUCTION
The OSGi framework implements a complete and dynamic component model in the sense of service-oriented computing. With OSGi applications or components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot [1]. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly. With respect to aspect-orientation, a couple of approaches supporting load-time or run-time weaving already exist. When it comes to integrating the aspect-oriented approaches with the OSGi platform, however, two issues arise. The first issue concerns the adaptation of aspect weaving to the restricted class visibility among bundles within OSGi. Since each OSGi bundle has its own class loader defined, there is no standard way for accessing classes that are located in different bundles. If a bundle wants to access services of another bundle, the bundle has to request the service registry. However, the bundle to be accessed at run-time has to be determined before compile-time. In that case, the OSGi registry returns a reference to the bundle that provides the appropriate service. After that, the requestor can submit a request to the bundle that was looked up in the registry. The bundle internal classes, however, are not visible to other bundles per se (e.g. if the required bundle packages are not explicitly exported and imported respectively). This raises problems when aspects have to be woven into a new bundle entering the system at run-time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Figure 1 OSGi Layered Architecture
Due to that restriction imposed by the platform, the connection between incoming bundles and the respective aspect bundles can be established during load-time the
earliest. That implies that we need to completely decouple aspect code from component code. Therefore, the second issue arising can be regarded as a consequence of the solution to the first issue. A mechanism needs to be provided for separating the weaving information from components that can be utilized during class loading.

The problem that we describe here can be mapped to the conceptual problem that has been already described by Griswold et al [4]: The authors state that it is the tight coupling between advices and join points in concrete implementations that raise the problem that the aspect developer needs to know the implementation, and therefore also has to rely on the stability of the implementation. We can conversely argue that we need to separate the advice from concrete pointcuts because we do not know all the component implementations beforehand. The solution proposed by the authors of [4] is to use so-called crosscutting programming interfaces (XPI). The principle is shown in Figure 2. The advice uses only the interface, or references to abstract pointcut definitions. That means in our context that during load-time the abstract pointcut is instantiated by concrete pointcuts, provided by the incoming bundle implementations.

![Figure 2 Separating Advice from Java Implementations (XPI)](image)

In our solution we utilize the ideas that are manifested in the XPI approach.

2. RELATED WORK

In the past, several efforts were made in order to support aspect-oriented concepts in dynamic environments. Typical examples are (a) JBOSS-AOP [9], (b) AspectWerkz [10], (c) Prose [11], or (d) AOSGi [7]. For checking if the existing approaches can be easily used with the OSGi platform, we first specify what concerns need to be covered: 1. In order to integrate existing approaches with the OSGi framework we need to be able to overcome the class-loading issue. 2. In order to do so, we need to decouple aspect definitions from concrete pointcuts as described in the previous section. That is, with respect to existing approaches that support load-time or run-time weaving, we need to investigate their capabilities in terms of separating advice from concrete pointcuts.

In general, JBOSS-AOP supports different binding times of aspects: compile-time, load-time and run-time. Concerning aspect definitions, JBOSS-AOP defines aspects in terms of interceptions. The interceptions can then be applied to specific parts of the implementation. However, JBOSS-AOP assumes the pointcuts to be defined before load-time, albeit the concrete advice implementation might be changeable during run-time.

AspectWerkz utilizes a sophisticated static weaving approach. In contrast to standard AspectJ, AspectWerkz introduces a layer of indirection. Hence, the advice do not have to be woven - they can be added, changed or removed at run-time. However, AspectWerkz assumes that the pointcuts are defined before run-time. Thus, the approach is not feasible to be applied in an OSGi environment without any changes.

Prose (Programmable Extensions of Services) defines aspects in a Java-like manner. Aspects defined in Prose can be used at run-time for adding or removing aspects already deployed. With Prose the developer is able to change the execution of methods. However, concerning other join point selections or other kinds of advice Prose has some limitations. Prose does not support introductions in terms of adding or replacing interfaces.

Concerning the integration of aspect-oriented concepts in the context of OSGi we found two existing efforts: The AOSGi project as well as an approach that aims at a similar solution as we do. That is, by changing the class loading strategy of OSGi bundles [12].

AOSGi aims at the integration of AspectJ load-time weaving features into Equinox, an implementation of the OSGi specification. At the moment version 1.0.4 of AOSGi is announced, which covers all required basic functionality. For instance, inter-bundle load-time weaving is supported, that is, classes can be woven even if the respective aspects are placed in different bundles. According to the AOSGi project, two models for applying aspects may be distinguished: "opt-in" and "co-opt". The difference between both models can be found in the "direction" of the weaving. Using the opt-in model the application developer has to be aware of those aspects that are supposed to be woven into the bundle’s classes. That is, the aspect-references have to be explicitly specified by means of Import-Package or Require-Bundle directives in a bundle’s manifest file. The co-opt model allows to specify aspect dependencies by means of an Eclipse-SupplementBundle, Eclipse-SupplementExporter or Eclipse-SupplementImporter entries. The last three additional directives provide a flexible mechanism of extending bundle dependencies without modifying the target bundles. Hence, AOSGi forces the developer to explicitly determine bundle dependencies to aspects via manifest entries. The aspects themselves can be bundled separately, however, their existence has to be known a priori. On the implementation level AOSGi exploits the extensibility of
the OSGi framework by using so-called extension bundles to bring forward load time weaving mechanism. It uses the hookable adaptor concept of Equinox, in particular, `AdaptorHook`, `ClassLoadingHook`, etc. interfaces as described in [8]. The hooks allow to intercept the class loading process and to perform adaptation if required. The weaver configuration occurs by means of AspectJ aop.xml files [7]. So looking for this file helps decide whether the bundle is an aspect bundle or not. The main disadvantage of both models is that there is no real support for separation of pointcut definitions and advices. In case of the opt-in approach the aspect dependencies between bundles exist at deployment time. So the existence of bundles at run-time must be guaranteed so that the bundles can be properly resolved and started. In case of the co-opt model the aspect implementer has to be familiar with the implementation details to define pointcuts correctly.

The authors of [12] change the class loading strategy of OSGi bundles to be able to access the classes that are defined within a so-called “Aspect-Manager” bundle. The strategy to access the system class loader of the OSGi framework, however, seems to compromise the principles of encapsulation and information hiding respectively. With our approach we will show that the class visibility can be customized in such a way that each bundle only gets access to the bundles' classloaders that are actually required for aspect weaving. Moreover, the authors do not present a concept for binding aspects to pointcuts during load-time.

3. APPROACH

Our approach comprises a solution to the class loading issue as well as to the load-time instantiation of the abstract pointcuts. To support a clear separation of advice and its referred pointcuts, we partitioned the aspect into two parts. The first part is the advice logic and the second part is the pointcut implementation. To that end, we have a bundle\(^1\) that is in charge of holding the advices, the so-called `AspectProvider`. All other bundles that are determined for weaving bring their specific pointcut implementation. At that point we stress the fact that the advice logic and the pointcut implementations are decoupled completely.

3.1 Class Loading

As stated earlier, the class visibility is restricted to the scope of the respective bundle, and therefore, the separation of advice and pointcut causes some problems, though. For that reason we developed a customized class loading concept for OSGi. Assuming the `AspectProvider` bundle running, we dynamically create a special class loader each time a new bundle is being started. This customized class loader (“Intermediate Class loader”) is initialized with a reference to the class loader of the AspectProvider bundle, and at the same time, as an extension of the newly entered bundle's class loader (see Figure 3). Using that construct, we exploit the java class loading mechanism by overriding specific methods, so the `Intermediate Class Loader` always finds the required classes. The resulting class loading process is then working as follows:

1. Apply the standard way the framework deal with class loading (`loadClass()`).
2. In case the class could not be found, the Intermediate class loader (ICL) is called (`findLoadedClass()`).
3. (4) Since the ICL can access the class loader of the Aspect Provider bundle, the aspect bundle can be checked for the required class.

The concrete implementation is explained in detail in the subsequent sections.

![Figure 3 Creating an “Intermediate class loader” as a connector to the aspect's classes](image)

3.2 The Weaving Process

To successfully cope with load-time aspect weaving two important challenges have to be solved:

1. Determination of the classes and the advices that need to be woven.
2. Support of loading dynamically constructed classes.

Extension mechanisms can be used for elegant coping with both of these challenges within the OSGi environment. The OSGi framework supports a mechanism to attach one bundle (called fragment) to another (called host). A fragment attached to the system bundle is called extension bundle. Extensions therefore can be used to insert additional functionality into the framework that is required for all running bundles (For more details refer to [1]).

\(^1\) In this example we use only one AspectProvider bundle, however, multiple AspectProvider bundles are possible.
Equinox implementation of OSGi provides a convenient way to implement and attach extension bundles. At that point, our concept exploits the same mechanisms as they are used in the AOSGi project: hookable adaptor. In particular, we use ClassLoadingHook as well as BundleWatcherHook for performing our weaving strategy. ClassLoadingHook provides functionalities to manipulate class loading via the method processClass (see step 2 in Figure 4). Hook objects are initialized by the adaptor during the launch of the framework [2].

3.2.1 Initialization Phase
The initialization phase is performed by an extension bundle called org.weaver.hookextension. It contains a class called Hook that implements ClassLoadingHook and BundleWatcher. We manipulate class loading by replacing the base class loader of all bundles by a class loader that is capable of resolving all classes within the AspectProvider bundle’s class space. To that end, Hook creates and sets an instance of ExtendedDefaultClassLoader (EDCL) as base class loader for all the bundles that are starting. EDCL enhances DefaultClassLoader and provides the class loading strategy described in 4.1. That is, EDCL possesses an instance of an ICL that can be used to load classes that are placed in the AspectProvider bundle. So we are able to intercept class loading of all bundles in order to launch class weaving before a class is defined by the EDCL. The EDCL performs standard class loading (step 1 in Figure 4) if the RuntimeWeaver bundle is not started. (Remark: In this case, the RuntimeWeaver bundle also contains the advice logic.)

3.2.2 Weaving Phase
Considering the RuntimeWeaver bundle running, the actual weaving phase is initialized in case a particular application bundle that has an aop.xml file defined tries to load a class (see Figure 5).

```
<aspect>
    <concrete-aspect name="C_Aspect" extends="">
    <pointcut name="simplepoint" expression="execution((interface.SimpleInterface.method))"/>
    </concrete-aspect>
</aspects>
```

Figure 5 Concrete pointcut definition
Please note that the extends-attribute of the pointcut is empty. By replacing the empty string with the name of the aspect that is supposed to be woven during load-time, we will be able to instantiate the abstract pointcut as it is used by the aspect bundle.

The bundle holds a reference to the ICL that is provided by Hook to EDCL invoking it to resolve classes in step 2 in Figure 4. The ICL accomplishes steps 3 and 4 in the presented class loading schema. The decision whether to start weaving a particular class can be made by looking at only those bundles that contain a "META-INF/aop.xml" representation of concrete aspects.

This mechanism also enables a dynamic match between concrete and abstract pointcuts (see Figure 6).

```
public abstract aspect A_Aspect {
    public pointcut simplepoint()
    {
        String around() : simplepoint()
        {
            System.out.println("Aspect woven");
            return "";
        }
    }
```

Figure 6 Aspect definition using an abstract pointcut

The RuntimeWeaver is responsible for the actual weaving. The bundle has a factory that is used by Hook to instantiate AJWeaver classes for all bundles having aop.xml files defined. AJWeaver matches all concrete and abstract pointcuts using their names and extends the concrete pointcut definitions of the respective bundles at load-time (see step 5 in Figure 4). That is, the extends-attribute as shown in the aop.xml in Figure 5 would be changed to extends="A_Aspect". After this step, the aspect advices are eventually “connected” to the concrete bundle pointcuts. Then, the AspectJ API can be used to fulfill weaving (see step 6 in Figure 4). The woven class is then passed back to EDCL for defining the class in the bundle’s scope.

The presented approach in this paper has been successfully applied in the context of architecture compliance checking. In that context we were able to instrument incoming OSGi bundles by using AspectJ. The instrumentation was used for collecting trace data that could be validated afterwards in terms of architectural compliance. The load-time weaving in the OSGi context enabled us to conduct
selective instrumentation of bundles to keep the profiling as well as the instrumentation code within acceptable limits.

4. SUMMARY

In this paper, we illustrated a light-weight approach for enabling advanced AOP capabilities in the context of OSGi. We motivated the work by showing the challenges of utilizing current aspect-oriented approaches in the context of the OSGi platform. We stress the fact that existing solutions already solve the conceptual issue of separating advice from concrete pointcut implementations, however, they might have to be adapted to become usable with the OSGi platform. In that sense, our load-time weaving approach might complement existing approaches like JBOSS-AOP in order to make it seamlessly work with OSGi.

For decoupling the pointcut implementation from advices we extended the class loading mechanism of java and exploited platform services for intercepting class loading. By doing so, we are finally able to connect advice and pointcut implementations during load-time. Eventually, our approach is not restricted to Equinox, potentially it can be used with other implementations of OSGi as well. The presented approach ca be considered “light-weight” because we do not presume any changes of OSGi or AspectJ specifications like in case of AOSGi, for instance.

Our approach relies on the determination of aspects through the aop.xml-files (similar to JBOSS-AOP) and makes the connection of pointcuts and advice dynamically at load-time.

5. REFERENCES

A Feature Model of an Aspect-Oriented Middleware Family for Pervasive Systems

Lidia Fuentes and Nadia Gamez
GISUM Group, Departamento de Lenguajes y Ciencias de la Computación,
Universidad de Málaga, Málaga (Spain)
{lff,nadia}@lcc.uma.es

Abstract
Pervasive applications will be naturally integrated as part of our environment. On the one hand, they will be deployed into a diversity of small devices and appliances, and on the other hand, they must be aware of highly changing execution contexts. Therefore pervasive computing requires the definition of advance mechanisms that support, (1) the deployment of pervasive applications through various devices with different capacities, and (2) the runtime reconfiguration for dealing with context changes. A software product line approach would be very useful to express the different requirements of devices in terms of commonalities and variabilities of a middleware platforms family. A feature model for pervasive applications will help both to deploy various configurations of the middleware tailored to each device, and to support the dynamic reconfiguration according to context. But, several crosscutting variable features and dependencies between features are commonly found in pervasive computing, (e.g. security, context-awareness, fault-tolerance, etc.). To address this problem we propose a family of aspect-oriented middleware platforms, able to deal with the high dynamic issue of pervasive systems. In this paper we will focus on the feature model definition and we also outline its mapping to a dynamic aspect-oriented middleware platform.

Keywords Feature Models, SPL, AOSD, Middleware, Pervasive Systems

1. Introduction
Pervasive computing is the next computational generation where the information and the communications will be available in a diversity of small devices and appliances, for every people and during all the time. Then, pervasive applications will be naturally integrated as part of an environment in constant change.

Therefore one of the challenges of pervasive computing is the definition of advance mechanisms that support the deployment and the tailored-configuration of pervasive applications through various kinds of devices with different capacities and characteristics. Another challenge is the provision of mechanisms that allow the runtime reconfiguration of pervasive applications for dealing with context changes. This means that a pervasive application have to deal with static and dynamic changes, so its architecture should be well modularized to facilitate its adaptation to the evolution of devices and environment.

A Software Product Line (SPL) approach would be very useful to express the different requirements of devices in terms of commonalities and variabilities of a middleware platforms family. Normally, a middleware platform provides all the common services most used by distributed applications. But in pervasive applications resource constraint is an important limitation, so only a specific middleware platform configuration that fits the device characteristics must be installed. Feature modeling analyzes commonality and variability from a domain perspective [6]. Then, a feature model allows specifying where is the variability in an independent way to the core asset, and enables reasoning about all the different possible configurations. A feature model for pervasive applications will help to specify different configurations of middleware according to existing devices profiles. This will allow the deployment of different versions of middleware tailored to the resource constraints of small devices and appliances. In addition, the explicit existence of a feature model specification will also help to support the dynamic reconfiguration of a middleware platform according to changing contexts, or to runtime variations of device resources. On the other hand, a software product line promotes the notion of architecture centric software engineering [1]. Then, a feature model can be used as input for generating an architectural representation of a product line.
Likewise most of current distributed applications, several crosscutting variable features and dependencies between features are commonly found in pervasive computing. Some of these crosscutting variables features are security, context-aware and fault-tolerance and so on. If we design these crosscutting features in a traditional way we will drastically reduce reusability, adaptability and the evolution of the product line.

In order to solve this problem of crosscutting variables features and dependencies between features we will use an Aspect-Oriented approach. Aspect-Oriented Software Development (AOSD) promotes the separation of concerns at every stage of the software lifecycle, from requirements and architectural design (early aspects) to implementation. Then, using this approach we can define the middleware architecture in a more cohesive and decouple way, alleviating the reconfiguration and the evolution tasks.

Normally aspect-oriented middleware, is understood as a middleware platform that supports the execution of aspect-oriented applications. But normally, the internal platform architecture of such middleware is not full aspect-oriented. Some efforts have been done towards the definition of a truly aspect-oriented middleware platform architecture, but normally only some specific middleware concerns are separated. In [4] some middleware specific concerns such as security or mobility are modelled as aspects. DyMac [5] is another aspect-oriented middleware platform, that provides an aspect-oriented definition of the middleware aspect weaver supporting advanced remote pointcuts, transparent remote advice and distributed instantiation scopes of aspects. Since the existing aspect-oriented middleware are not fully aspect-oriented, we propose a family of aspect-oriented middleware platforms, able to enable, disable or replace the version of some services modelled as aspects depending of the available resources of small devices. Our approach also include a mapping between the middleware feature model and an aspect-oriented architecture. Likewise, other previous proposals [7], this platform will support the dynamic composition driven by an aspect-oriented Domain Specific Language (DSL) defined for this purpose. This DSL will be an extension of an aspect oriented Architecture Description Language (ADL) with variability. In this paper we will focus on the feature model definition and we also outline its mapping to a dynamic aspect-oriented middleware platform.

After this introduction, this paper is structured as follows. In Section 2 we present an overview of the requirements of pervasive system, SPL and feature models and concepts of aspect-oriented middlewares. In Section 3 the feature model of our middleware is detailed. In Section 4 we outline the mapping between the feature model an one architectural representation. Section 5 comments on related work, and finally, Section 6 outlines some conclusions and future work.

2. Background

2.1 Pervasive Systems Requirements

Pervasive computing is becoming a reality. In pervasive systems we can found a great diversity of computing facilities (computers, PDAs, sensors and so on) and high diversity of networks technologies (mobiles ad-hoc networks, sensors/actuators, ...). Then is a big problem how to tackle these high diversity environments.

Regarding this issue we can resume the main problems of pervasive systems in:

- Hardware heterogeneity: the embedded systems and mobiles devices have different capacities and constraints, such as the amount of available memory, communications kinds or computations capacity.
- Dynamism of the application environments: the pervasive system has to be able to react in an automatic way to the environment changes, i.e. they must support a dynamic auto-adaptation and reconfiguration.
- Management of the application evolution: hardware and software technologies in pervasive system are evolving and changing continuously, then these systems have to be easily to maintain and evolve.

Middleware can be the underlying technology to address these troubles. In order to solve these problems the middleware has to provide mechanisms for tailoring the configurations regarding the devices. It has to support also the context changes and adaptation itself to this changes. And finally it has to allow the reuse in order to enable the software and hardware evolution.

The middleware for pervasive system has special requirements beside of typical platforms services. It has to allow to reconfigure itself in design time and in runtime. For example, it has to be able to enable or disable some services or replace the version of one service depending of available resource of the devices. Then, many of the middleware services will be variable, optional when they can be running or not in one specific configuration, or alternative if they can be implemented in one way or in other different way.

2.2 Software Product Lines and Feature Models

Product Line Software Engineering (PLSE) have the ability to exploit commonality and manage variability among products from a domain perspective [6]. In the feature-oriented approach, commonalities and variabilities are analyzed in terms of features. A feature is any prominent and distinctive concept or characteristic that is visible to various stakeholders [6]. The features can be organized into a feature model that represents all possible products of a software product line. Feature modeling analyzes commonality and variability from a domain perspective. Commonalities are modeled as mandatory features and variabilities are modeled as vari-
able features which are classified as alternative or optional features.

In features diagrams three kinds of relationships are found: compose of (when the feature is composed of several sub-features), generalization or specialization (when the feature is a generalization of the sub-features), and implemented by (when the sub-feature is needed to implement the feature). Furthermore, for each variable feature, feature dependency analysis can identify dependencies between features. Examples of such dependencies are the mutual dependency and mutual exclusion relationships. Finally the features can be classified in four layers: capability, operating environment, domain technology and implementation technique. The capability layer is composed by the user visible characteristics, such as services, operations, non-functional characteristics and so on. The operating environment layer contains the environment in which the application is used. The domain technology layer are the way to implements the capability layer elements. Finally, in the implementation techniques layer are the techniques used to implements the capability layer elements.

2.3 Middlewares for Aspect-Oriented Applications

As we mentioned in the introduction, many of the proposals that claim to provide an aspect-oriented middleware level, are really non fully aspect-oriented platforms for developing aspect-oriented applications. Our proposals tries to bring the aspect-oriented benefits to the middleware itself, in order to deal with heterogeneity of devices and communication technologies and the high dynamism of pervasive environments. Considering this approach, the final application will be just another platform component. This means, that the specific functionality of final pervasive applications could be developed using a non aspect-oriented language. This will encourage developers of pervasive applications, usually non experts in aspect-orientation, and more accustomed to low level programming to use our middleware.

In an AO approach the middleware can be considered as a collection of aspects with many of them being completely standard (formerly known as common services such as persistence or security). Normally in non-AO middleware the list of common services offered by a platform is fixed, and is not possible neither extend nor customize them by end users. The aspects provide us the facility of add, remove or change an architectural element without modifying the rest of the architectural. In AO middleware the end user may add new services in a non-invasive way, by means of adding a new aspect, for adapting the platform for example to a new application domain. So, in an AO middleware is permitted to define extensions of an AO middleware that can be highly proprietary, developed by end users. Let’s consider the definition of a custom authentication. Using an AO approach end users can decide whether to use or not a this security aspect depending on the execution environment. This can be realized as simple as, for secure environments (e.g. inside an Intranet) the security aspect is not considered as part of the aspect composition rules, but for insecure environments yes. In the latter case, the first user access will be intercepted and an authentication aspect will be evaluated after message delivery. Even more, since the security aspects are modelled separately from application components is easier to change their implementation and use for example a user and password authentication or if the device has a intelligent card reader use this method to authenticate. Furthermore, the implementation can be changed at runtime if for example the card reader ar plug-in during the application execution.

CAM/DAOP [7] is one of the well-known platforms for developing aspect-oriented applications, and one of our most relevant previous works. One of its most significant contributions is the dynamic weaving of components and aspects even at runtime. In this work we will follow a similar approach about the dynamic weaving.

3. Feature Model of the Middleware

Platform Family for Pervasive System

In this section is shown the feature model of the middleware. In order to design a correct feature model we have follow the guidelines proposed in [6], as we mentioned in the background section.

Our middleware will follow a microkernel and services structure. The microkernel term describes a form of operating system design in which the amount of code that must be executed in privileged mode is maintained to an absolute minimum [4]. As a consequence, the rest of services are built as independent modules that are plugged and executed by the kernel. In this way, we obtain a more modular and reusable system. Furthermore, we distinguish between mandatory base services and the rest of optional extra services that will be added according to applications requirements.

Then, as is shown in Figure 1, the middleware feature is composed by two mandatory features, the microkernel and the services that can be basic or extra services. The basic services is also a mandatory feature but the extra services is an optional feature. All these features are in the capability layer, this layer is more detailed in next figures. As is shown in the background section, the capability layer contains all the user visible characteristics. In our case the user of the middleware will be the applications, then in this layer we found all the microkernel details and all the services.

The middleware can be used in many kinds of devices (such us, PDAs or mobiles phones) and allow to use several communication types (for example, wifi, bluetooth). Then, in the operation environment layer that represents the environment in which is used the application, we place these two features (device type and communication type). The device type feature will have many alternatives sub-features depending on the kind of devices. The same thing happen with the communication type feature. The domain technol-
Then the optional security service will be active only in these moments. Secondly, the environment can change and some services may not be executed in the same way that before, for example if one device is missing. Thirdly, the pervasive system has to be aware of device constraints, e.g. if the battery is less than a low value maybe all the graphics in the application must be changed by text. Finally, user preferences also provokes changes in the context. registered the device constraint, optionally the user preferences and other context properties. The last sub-feature of the microkernel is the service manager that manages the architecture of the middleware and runs the application. It has two sub-features, the architectural manager and the weaver. The last is an interpreter that composes all the elements of the middleware and application. Like our middleware follows an aspect oriented approach we call this interpreter as weaver. In following sections is detailed its functioning. The architectural manager is composed by the architectural description and the selector. The architectural description is composed by two sub-features, the middleware product line architecture description and one instance of this product line that represents the architectural description of a particular product of this middleware product line. The selector is a tool that produces a particular product from a product line, taken account the context properties (device constraints, user preferences, etcetera).

In Figure 3 is shown the base services feature model. All these services are mandatory feature. The lookup service search where the middleware services are running. The discovery service search for other devices that are using the middleware. The communication implements communication between all the middleware elements. The device manager service controls the devices where the application is deployed. Finally, fault tolerance is used when the middleware elements lose the connection.

On the other hand, the extras services are shown in Figure 4 and all are optional. These services are facilities that the middleware gives to the application. The security service provides a secure environment to the application. This feature is composed by many alternative sub-features, for example encryption or authentication. Persistence is other traditional services that allows to store the data in several places in order to avoid loss. The error handling services provide several kind of protocols that the application can use to handle an error. The context-aware services is the responsible of be aware of any change in the context (as we said be-
fore, changes in application, environment, devices or user). Finally the position location is a service that allows to deal with the different location technologies and data representation.

4. Mapping the Feature Model to an AO Middleware

4.1 Mapping the Feature Model into a Product Line Architecture

As is mentioned before, software product line promotes the notion of architecture centric software engineering [1]. On the other hand, our middleware will be driven by architecture. Then, seem to be suitable generate the architectural representation of the middleware product line from the middleware feature model.

In [3] we present a mapping between feature models and an aspect-oriented architectural description, specifically an extension of xADL [2] with aspects. This extension shows a symmetric decomposition model where a component is considered an aspect when it participates in an aspectual interaction. These interactions are placed in the connectors. Then, we call ‘aspectual’ components when a component participate in an aspectual interaction, and we call ‘aspectual’ connector when the connector express crosscutting relations. In [3] the ‘aspectual’ component is used not only for crosscutting features, but also for solve the dependencies between features.

In our architecture, all the middleware services are crosscutting features, then will be mapped into aspectual components. The only component in the middleware will be the microkernel. And also the application will be considered like other regular component.

As mapping example we can consider the context-aware service, that will be modeled as an ‘aspectual’ component. Then, there will be an ‘aspectual’ connector that specify when that component has to be executed. The context-aware service has to do something when for example the battery of one device is less than a critical value. In such case, this service has to call the context manager microkernel component, and this has to call the selector in order to create a new middleware architecture that deal with that battery restriction. Then, for example there will be an ‘aspectual’ connector that specifies that when the battery sends a message with its current value to the monitoring the context-aware service has to check the vale and take into account. This abbreviated XML fragment represents the context ‘aspectual’ connector (some tags are omitted for the sake of clarity):

```xml
<types:connectorType types:id="AspConnType">
  <sendMessage>
    <sourceComp>
      <roles>Battery</roles>
    </sourceComp>
    <targetComp>
      <roles>Monitoring</roles>
    </targetComp>
    <targetMessages>
      <message name="valueBattery"/>
    </targetMessages>
  </sendMessage>
  <AFTER_SEND>
    <concurrent>
      <aspect id="context-aware"/>
    </concurrent>
  </AFTER_SEND>
</types:connectorType>
```

In our approach we will do a similar thing but using a DSL that we will define. As our middleware will be driven by architecture, the architectural representation made with this DSL will be taken as input to the weaver in order to compose the middleware elements. This composition is made at runtime and changing the architectural description the middleware will change also, allowing in this way the dynamic reconfiguration of the middleware.

4.2 AO Middleware Architecture

As is shown in the previous section the middleware architecture will be composed by two components (the application itself and the microkernel) and several aspectual components (all the services). Likewise most of traditional middleware platforms architecture, we follow a layer approach (Figure 5). The application level is in the top and the middleware itself is just after. The first sub-layer of the middleware are the application services like security, context-awareness, error-handling, and so on. These services use the base services that are in the next layer, such as communication, lookup, etcetera. Below these layers is the microkernel, that, like is shown in the feature model description, is composed by the container, the factory, the context manager and the service manager.

The product line architectural description of the middleware is place inside of the service manager. The microkernel, through the context manager, has to know the context properties. With this information and with the architectural description of the application, the service manager runs the
automatic selector facility in order to instantiate a particular middleware architecture of the product line architecture. This is a static configuration of the middleware.

Using the context properties, the factory will instantiate the specific selected services. In order to run the application the service manager has the weaver that is the responsible of make the composition between aspectual components, corresponding to the services, and the components (microkernel and application).

If a change is made in the context (user properties, device constraints, ...) the microkernel will use the selector again and a new particular middleware of the product line will be instantiate. The responsible to perform this is the extra service context-aware that has access to the context manager. Then, this is the dynamic reconfiguration of the middleware.

5. Conclusions and Future Work

In this paper we have shown the troubles of the pervasive systems and how a middleware approach can solve them. We have follow the layer structure for the middleware, with a microkernel and several services in order to get a more reusable and adaptable middleware. Furthermore, we have proposed to use an aspect oriented middleware that allows dynamic reconfiguration, because the reconfiguration is an issue very important for context-aware pervasive system. Mainly, in this paper we have focus in the design of the feature model of the middleware product family, that after it will be used to produce the middleware architectural product line. Then, also we have outlined a mapping between the feature model and the architecture and we have shown the architecture of the middleware following the layer approach.

This is the initial phase of the middleware development then as a future work we have to continue designing the aspect-oriented DSL. Then, a mapping between the feature model and architectural representation in the DSL will be done. After, having the product line architecture we will start the implementation of the middleware, beginning with the microkernel and after with the services. During all this process new services of the middleware product line can appear as needed, then the product line feature model will have modified.

Acknowledgments

This work has been supported by Spanish Ministerio de Ciencia y Tecnología (MCYT) Project TIN2005-09405-C02-01 and European Commission Grant IST-2-004349-NOE AOSD-Europe and the European Commission STREP Project AMPL-IST-033710.

References

Building a Distributed AOP Middleware for Large Scale Systems

Rubén Mondéjar, Pedro García, and Carles Pairot
Departments of Computer Science and Mathematics
Universitat Rovira i Virgili
Tarragona, Spain
{ruben.mondejar, pedro.garcia, carles.pairot}@urv.cat

Antonio F. Gómez Skarmeta
Universidad de Murcia
Department of Computer Engineering
Murcia, Spain
skarmeta@dif.um.es

Abstract

Building large scale applications is nowadays a complex challenge. Such complexity is determined by several factors like distributed application development, deployment or management, to name a few. Adaptive middleware plays an important role in achieving such task, and abstracts developers from the underlying layer issues like persistence, fault tolerance, and load balancing, among others. Distributed Aspect Oriented Programming (AOP) is a promising paradigm that offers new ideas in the middleware arena. Several models like remote pointcut or component-aspects for designing wide-area distributed systems exist in such setting, but none of them completely fulfill the scalability requirement. In this paper we present a distributed aspect middleware for large-scale systems mainly offering three contributions. Firstly we introduce a complete aspect remoting service with one-to-one and one-to-many abstractions. Secondly, we outline the construction of a distributed aspect meta-model that provides a novel distributed meta-pointcut mechanism to intercept remote services. Finally, the distributed aspect composition model that allows connection mechanisms in design, activation, and runtime phases. The last part of the paper includes a proof-of-concept, consisting of an adaptive Distributed Hash Table (DHT), which is a clear example of how a wide variety of distributed aspects on large-scale scenarios can be implemented by using our model.

Categories and Subject Descriptors D.2.11 [Software Engineering]: Software Architectures, C.2.4 [Computer-Communication Networks]: Distributed Systems, J.8 [Internet Applications]: Middleware

General Terms Design, Performance.

Keywords distributed AOP, Peer-to-Peer, adaptive middleware.

1. Introduction

The continuous advances in adaptive middleware architectures have been changing the way distributed systems are being developed. Particularly, several approaches have been proposed for building software that can dynamically adapt to its environment. In addition to this, it is desirable that adaptive systems can provide adaptation to achieve new goals that were unforeseen during design or, even runtime phase.

In such arena, a new paradigm called distributed AOP has recently appeared. Distributed AOP defines many new concepts like remote pointcuts [1], which are similar to traditional remote method calls, since execution is implied on a remote host; component-aspects [2,8], which introduces some features from component models, and aspect group notions [3], thus establishing a context where aspects can be deployed in a set of hosts.

However, we can find three major drawbacks in these current distributed AOP works. First, some of them suffer from limitations when adapted to large-scale environments. Second, the group notion is essential for some kinds of distributed applications but only a few approaches are aware of it. Finally, the common distributed AOP test is "if the remote pointcut abstraction is available, then it is a distributed AOP framework". Nevertheless, we believe that a more formal definition is needed so as to be considered a new distributed paradigm comparable to remote objects or distributed components.

On the other hand, in large-scale scenarios, peer-to-peer (p2p) networks gradually emerged as an alternative to traditional client-server systems for some distributed domains. In this way, and by making good use of p2p mechanisms we have designed Damon [5]. Damon is a fully decentralized aspect deployment platform built on top of a structured p2p overlay network. It provides the necessary services to deploy aspects in a large-scale network, like persistence, runtime activation, or a messaging system.

In order to solve the named issues of the distributed AOP area, we present a distributed aspect middleware over large-scale networks. In this sense, the three main contributions of our model are as follows:

• Mapping of large-scale abstractions for aspect remoting, namely one-to-one and one-to-many, designed for aspect activation, remote pointcut and remote invocation.
• Design of a distributed aspect meta-model, using a new mechanism: the distributed meta-pointcut.
• Provision of aspect composition capabilities in runtime, based on abstract descriptors to represent distributed aspect properties, like aspect remoting services.

The rest of the paper is structured as follows. In Section 2 we give a brief overview of Damon’s distributed aspect deployment platform [5]. Section 3 describes our distributed aspect middleware, and all the services that it offers. In Section 4 we analyze a possible usage scenario, and in Section 5 we explore related work. Finally, we conclude by providing an outline of future work.
2. Damon Overview

The principle of separation of concerns suggests a problem where a number of concerns should be identified and separated. In this way, we are mostly interested in crosscutting concerns where aspects are executed in a remote host or simultaneously in multiple hosts. Examples of these emerging concerns are fault-tolerance, load-balancing, replication, synchronization, among others.

In order to implement these decentralized crosscutting concerns in large-scale systems we developed Damon [5]. Our platform offers the necessary mechanisms to deploy and execute aspects in a p2p network. The main features of Damon include:

- The definition of a Decentralized Aspect Container offering:
  - The use of p2p URL locators (i.e. p2p://aspect.sample) which dynamically allow mappings of aspects, hosts and groups of hosts.
  - A Decentralized Aspect Naming service, which provides the necessary infrastructure to deploy and locate aspects in a decentralized and distributed way.
  - An Activation mechanism, which notifies remote destination host(s) to install aspect(s) and activate them locally.
- A set of mapped underlying functionalities, including:
  - A Routing system, which benefits from the underlying p2p network layer to perform Key-Based Routing (KBR).
  - A Messaging system that maps a set of one-to-many communication primitives on an application-level multicast (CAST) service.
  - A Persistence service, thus allowing aspects to store and retrieve information to/from the network.
  - A Reflective layer, which benefits from the host and network data instrumentation layer.

For more information about KBR and CAST systems, please refer to [6].

![Figure 1. Distributed Aspect Layered Architecture.](image)

To better understand how Damon works, we briefly describe the different phases of an aspect’s deployment and activation, and how Damon provides transparent services to handle them correctly. First, the aspect is coded to run on Damon; next, it is inserted in the platform with an assigned p2p locator and making it available to any host. Whenever the aspect is needed, it has to be located using its p2p locator to identify the destination host or the group of hosts. Finally, when the aspect is activated, it can use the underlying functionalities according to its behaviour(s) and/or requirement(s).

By using the PlanetLab testbed [5,7], we verified that Damon does not impose an excessive deployment and activation time overhead. Furthermore, we have also developed some solutions over Damon, like providing load-balancing in web server clustering [7]. Finally, we have also constructed a set of solutions that become the building block of a new model for distributed aspects, which we present in the following section.

3. Distributed Aspect Middleware

In this section, we present the three main contributions of our work in the distributed AOP middleware. Our purpose is to define a complex model which improves the design of distributed aspects in large-scale systems.

In Figure 1 we can observe the underlying services of Damon deployment aspect platform, explained in the previous section, and the new layer of our distributed aspect middleware. The rest of this section describes each part of our middleware architecture thus focusing in the distributed aspect entity, the aspect remoting services, the meta-aspect model, and the distributed composition model.

![Figure 2. Distributed Aspect Diagram.](image)

Indeed, our model is inspired in different distributed disciplines, like connection oriented or composition models, and mimics some characteristics from the CORBA Component Model (CCM) [10]. However, the first differential fact between these models and ours is our main entity, the distributed aspect. We define a distributed aspect in Figure 2, and we explain each of its elements as follows:

- **Source Hook**: aspects do not require the use of remote interfaces. As a consequence, distributed AOP provides a more versatile contract, since we can just invoke any of the entity’s functionalities only establishing the necessary hooks (local pointcuts).
- **Remote Pointcut**: it is a service for identifying source hooks, and remotely propagating associated join points. Distributed aspects can disseminate pointcuts to a single host or a group of them. Therefore, distributed aspects can work together without being tightly linked, thanks to the use of the connection model mechanisms. Think of the propagation of a sensor’s alarm to one or a set of remote actuators whenever each sensor’s source hook is triggered.
- **Remote Advice**: distributed aspects can be linked themselves together by means of a remote advice. The receptacle allows a distributed aspect to declare its dependency to a concrete remote pointcut. Thus, a remote advice uses the same identifier to be notified.
about its desired remote pointcut. Keeping in mind the previous example, actuators are to be subscribed via remote advice to the sensors’ remote pointcut.

- **Remote Invocations/Methods:** traditional method invocation is also supported on our model, and it serves a main purpose of inter-aspect communication on demand. If the actuator needs some specific information on a sensor, it can remotely inquire the sensor aspect via a remote invocation.

- **Shared Data:** it allows stateful distributed aspects to save/restore their state information. Moreover, in group aspects, this state can be shared by one, many or all members of the group.

Now that we have already defined the concept of what a distributed aspect consists of, we pass on to explain our model’s features, enumerated as follows:

- The Aspect Remoting Concept, including
  - Remote Services
  - Remote Abstractions
- The Distributed Meta-Aspect Model, defining
  - Distributed Meta-Pointcut
- The Distributed Aspect Composition
  - Distributed Abstract Descriptor

Upcoming sections explain these kinds of services, and the connection mechanisms in our composition model.

### 3.1 Aspect Remoting

We classify remote services into three categories: activation and passivation; remote pointcuts and advices; and remote methods and invocations. These services are what we define as the aspect remoting concept. Nevertheless, the activation and passivation services are to be provided by the underlying distributed aspect container. All these aspect remoting services use asynchronous/synchronous communication mechanisms performed by means of underlying messaging and routing capabilities.

As we can see in recent distributed AOP works [1,2,3,4,8], remote pointcuts and remote advices are nowadays the differential fact for this kind of frameworks, and obviously they become a fundamental part of our model. Moreover, most of these frameworks are based on traditional remote objects. Thus, they use remote invocations to perform remote pointcuts, but none of them offers the possibility of using remote invocations between distributed aspects.

In order to provide more flexibility, we consider the inclusion of remote method invocations as a key fact to provide efficient inter-aspect communication on demand. Distributed aspects are scattered among the network, and they need to execute their own methods, as found in traditional AOP, but in a distributed way. In such scenario, methods are to be dynamically invoked on other hosts. If we do not have remote invocations, we are forced to use remote pointcuts, which is confusing.

Regarding the variety of scenarios aspect remoting services have to deal with, we have defined a set of communication abstractions which are presented in the next section.

#### 3.1.1 Abstractions

Using the scalable nature of the underlying infrastructure, we implement several abstractions that provide a mapping from a set of large-scale communication services into our distributed middleware. We divide them into two groups: one-to-one and one-to-many abstractions.

**One-to-one abstractions** can be synchronous or asynchronous, depending on whether a developer wishes to block the originator execution until a result is returned.

The capabilities provided by the routing substrate are the foundation for what we call the one-to-one **hopped** abstraction. The advantage of using a key to route messages to the host is that we do not know anything about the destination. Moreover, when the host we are using goes down, the message would automatically route to another host, in a transparent process, and the originator continues to use the same key to route messages. However, hopped abstractions are not as efficient as **direct** abstractions. Depending on the used routing substrate, this approach incurs additional overhead, because a message is routed by one or more hosts reaching its destination. This philosophy remains in stark contrast to that of abstraction calls, where the message is moved directly from source to destination. In one-to-one direct abstractions we need to know the destination host’s URL locator, or directly its real IP address.

On the other hand, **one-to-group abstractions** are modelled using the messaging system layer by means of disseminating events. Moreover, thanks to the underlying p2p network, these abstractions benefit from a proximity-based topology. To target the destination group we also use p2p URL locators, i.e. p2p://group.urv.cat.

The any abstraction is an interesting way to provide remote services that benefit from network locality. If the messaging layer provides us with an efficient anycast primitive, we can use it to create a call to the distributed aspects that belong to the same group. The originator is insensitive to which group aspect provides data; it only wants its request to be served. The idea is to iterate the group members, starting from the closest member in the network. Once a member of the group is found to satisfy the condition, it returns an affirmative result. The many abstraction is a variation of the any abstraction. It takes advantage of the many abstraction provided by the messaging layer and it therefore sends a message to several group members, continuing to route until it finds enough members to satisfy a global condition. Similar to the any abstraction, when a distributed aspect receives an invocation, it first checks whether it satisfies a local condition and, subsequently, checks whether a global condition is met. The many abstraction is successful when the global condition is met. Finally, the multi abstraction is a remote invocation from a distributed aspect to all members of a group (for example, to propagate some information).

#### 3.2 Distributed Meta-Aspect

Currently, a meta-aspect is defined by [11] like “an aspect of an aspect, or of another meta-aspect. The meta-aspects operate on the aspects, being higher-order aspects”. In such way, we have designed a new meta-aspect concept but in a distributed manner which allows transparent interception of aspect remoting services. Therefore, we introduce this idea in our distributed middleware and we create the distributed meta-pointcut mechanism inside of the distributed meta-aspect entity.

In this line, our distributed meta-pointcut performs the remote interception by specifying a remote pointcut or invocation as its source hook. Moreover, the interception can occur in different moments of the remote service execution:

- **Before:** it is performed on the host where the remote service is originated. For example, if the sensor’s alarm is propagated in p2p://sensor.net group, a distributed meta-pointcut can listen to this alarm in the moment that it is generated.
- **After:** when the remote service gets to the destination host(s), the distributed meta-pointcut is invoked. This case is the opposite of the previous one, and it occurs just before remote advice or method execution. More-
over, it has access to the reflection information from the remote service. In the sensor’s example, this can intercept the actuators instead of sensors, and retrieves information about the propagation.

- **Around**: in such case, the interception is raised on any of the travelling hosts between the originator (before) and the destination (after). It is the most complex case, because we need to filter the traffic in the intermediate hosts and analyze the transient messages. In addition, this distributed meta-pointcut allows blocking and cancelling of the original service routing.

Furthermore, our distributed meta-aspect can optionally modify the reflection information provided by the remote services (around and after), like the address of the originator host, or the number of travelled hosts.

An important observation is the fact that in order to implement a distributed meta-pointcut we need mechanisms for intercept remote service on the distributed AOP frameworks. Moreover, these necessary mechanisms must be provided by the framework. We think that it is an approach with a low computational cost, and that remains fully transparent for the developer.

We consider that our distributed meta-model can be especially useful for distributed aspect design. Some use case scenarios are: monitoring, remote pointcut connection (aspect composition, or design patterns), propagation to other groups, or amplification or reduction of the service abstraction scope (i.e. any to many).

In Section 4, we present a proof-of-concept for our model, which includes two distributed meta-aspects (Monitor and Caching), using both of them as a meta-pointcut mechanism to behave accordingly.

### 3.3 Distributed Aspect Composition

In this section, we present the upper level contribution, our composition model. In order to abstract the previous mechanisms, we are willing to represent distributed aspects in a graphical diagram showing their properties and connectors (Figure 2). To achieve so, we generate an abstract descriptor (using XML language) that allows designers to deploy and change the aspect behaviour maintaining the abstraction level from the underlying implementation. Such XML descriptor defines the distributed aspect properties, the offered services that are propagated, and the remote required services which must be triggered.

Following this principle, composition means a conceptual abstraction for entity relations. Therefore, we provide a distributed aspect composition based on remote connectors. Component-aspects [8] suppose an initial approach that uses component composition. In addition, we present a dynamic and distributed connector model based on aspect remoting and meta-model concepts. Furthermore, we have a three phase composition process: design, starting with the diagram (i.e. Figure 3), and next generating the descriptors, or directly defining them; activation, using the descriptor to install the associated distributed aspects, and defining their behavior and connections; and, specially, in runtime, that allows redefinition of the activated distributed aspects, without redeploying any of them.

As a consequence, the distributed aspect’s XML descriptor structure includes aspect information to abstract the underlying implementation. Moreover, only the source hook’s specification requires specifying the method’s (local advice) name. This way, the pointcut’s declaration, which makes use of the AOP language, remains as transparent as possible.

In order to clarify this point, we include a descriptor example based on a distributed aspect of our proof-of-concept (Section 4) as follows:

```xml
<distributed-aspect>
  <name>Locator</name>
  <abstraction>MULTI</abstraction>
  <target>p2p://dht.urv.cat</target>
  <state>Stateless</state>
  <offered>
    <name>hashtableHook</name>
    <type>remote-pointcut</type>
    <id>locate</id>
    <required>
      <remote-method>hashtableHook</remote-method>
      <name>getValue</name>
    </required>
  </offered>
  <required>
    <name>valueArrive</name>
    <type>remote-method</type>
    <id>getValue</id>
  </required>
</distributed-aspect>
```

As we can see, the descriptor has three differentiated parts: distributed aspect properties, the offered, and the required remote services. Concretely, this distributed aspect propagates an asynchronous remote pointcut, named `locate`, by using the hopped abstraction from the `hashtableHook(..)` source hook method. Also, it has the `valueArrive(..)` remote method listening to `getValue` remote invocations.

On the other hand, we can define distributed meta-pointcuts in our descriptor definition, at the same level of offered and required properties. Such an example is shown as follows:

```xml
<distributed-aspect>
  <name>Caching</name>
  (...)
  <meta-pointcut>
    <name>locateInterceptor</name>
    <id>locate</id>
    <type>around</type>
    <target>p2p://dht.urv.cat</target>
  </meta-pointcut>
  (...)
</distributed-aspect>
```

Thereby, this `around` distributed meta-pointcut intercepts the `locate` remote service when it is routed to its destination host.
4. Proof-of-Concept : Adaptive DHT

The main scenario where we can use our middleware is to develop a large-scale distributed application. Therefore, this section provides a sample use case for our distributed aspect middleware proposal. We emphasize how the services provided by our model are used to create this new application. The chosen large-scale application is a Distributed Hash Table (DHT) [6] hooked to a traditional local hash table implementation.

The DHT application provides the same functionality as a traditional hash table, by storing the mapping between a key and a value. This interface implements store and retrieve functionalities, where the value is always stored at the active host(s) to which the key is mapped by the routing layer. In addition, it can help us to understand the message flow implied for the remote services.

Indeed, this sample scenario stresses how to construct complex distributed applications in a modular and transparent way. We can see this in Figure 3, which shows an example of complex interactions on the p2p://dht.urv.cat group.

For this purpose, we have implemented several decentralized crosscutting concerns like replication, location, fault-tolerance, load balancing, or monitoring. In such line, integration with any application is very simple. The rest of this section describes the DHT’s execution step by step as shown in Figure 3, focusing on the services provided by our distributed AOP model.

Step 1A. The starting point of this application is the Source Hook of the Locator distributed aspect. In addition, these distributed aspects are directly connected to the java.util.Hashtable class. The Locator aspect is deployed and activated on all members of the p2p://dht.urv.cat group. The main purpose of this distributed aspect is to intercept Hashtable’s main methods (get, put, and remove) and their local execution.

Step 1B. These executions are propagated as a remote pointcut. Consequently, the remote pointcut is routed host to host to the key’s owner node, by using the hopped abstraction. Once the key-value pair has reached destination, the registered remote advice is triggered and the Replicator distributed aspect is activated on the owner’s host. This distributed aspect has already been activated at startup on all members of the p2p://dht.urv.cat group.

Step 1C. When dealing with the put use case, we pretend to avoid any data storage problems which may be present in such dynamic environments as large-scale networks. Therefore, data is not only stored on the owner’s node, because if this host leaves the network for any reason, its data disappears. In order to solve this issue, information is replicated in some Replicator instances. Therefore, the owner’s running Replicator propagates the copy to other instances by using a remote invocation with the many abstraction.

Step 1D. Finally, when dealing with the get or remove use cases, the Locator receives the obtained value by means of a remote method call triggered by the Replicator owner.

Once we have the application running, then we can add new functionalities as well. In this sense, we can introduce the distributed meta-aspect facility in order to extend or modify the current application’s behaviour. More specifically, these distributed meta-aspects create a feedback control over the application, obtaining introspection information, and using it in order to improve, for example, the value request technique, as shown right away.

Step 2A. We activate the Monitor distributed meta-aspect in order to observe remote service activity, and to store information about these interactions. Thereby, the Monitor scores all of the host’s traffic, focusing on the key-value insertions. For this purpose, it introduced an after distributed meta-pointcut that intercepts Replicator’s requests (Step 1B) from the Locators in a transparent way.

In this line, the Monitor is thought to be activated on any or many host(s) depending on the group size. Therefore, by using the any abstraction for the previous meta-pointcut, the closest instance receives the information, thus avoiding that any Monitor instance becomes a bottleneck.

Step 2B. The idea is that the Monitor activates instances of the Caching distributed meta-aspect on the most transited hosts, using its own generated reports about traffic on key-value routing paths.

Step 3A. The Caching owns an around distributed meta-pointcut that intercepts the remote pointcut from Locator to Replicator distributed aspects (Step 1B). Subsequently, Caching obtains the value insertions that travel for its host. In this manner, a temporary copy can stay in cache during a specific time frame.

Step 3B. Finally, the remote meta-pointcut can decide if the original remote pointcut can continue to be routed or not. Thereby, it is here where it verifies if the key of a request has a cached value available. If this query is satisfactory then the remote pointcut (Step 1B) stops being routed and the result is sent back directly via remote invocation, like in Step 1D.
5. Related Work

Indeed, most of the work to date in distributed AOP [1,2,4,8] has been based on remote method invocation using a remote object framework, like RMI [http://java.sun.com/products/jdk/rmi/]. This fact limits these works to provide only one-to-one primitives, and makes the construction of group services even harder. Exceptionally, AWED [3] is based on a message oriented middleware (JGroups [http://www.jgroups.org]). Therefore, its network substrate allows one-to-many services, and provides the group abstraction.

However, although some of these works express that can be suitable for large-scale systems, this cannot be possible, because none of them can scale due to its underlying infrastructure. Anyway, we think that most of them have functionalities or mechanisms that can be included in our model like state sharing [3], distributed control-flow [4], or the distributed joinpoint infrastructure [8].

Previous to [1] related solutions are not considered distributed AOP, since they use traditional AOP in distributed systems but in a local way. On the other hand, it is of particular interest the use of AOP into the component-based software development (CBSD) discipline, which tries to settle the crosscutting concerns for design and development of reusable software component-based distributed applications. One example is CAM/DAOP [9], a component-aspect model that weaves aspects into components in runtime, following the AOSD perspective of a CBSD framework.

6. Conclusions and Future Work

In this paper we have presented a distributed aspect middleware built on top of a large-scale network. By using the previous Damon deployment platform, our model manages to provide many interesting mechanisms to distributed AOP developers. Our main contributions include aspect remoting services, a distributed meta-aspect model, and a composition model, providing a set of necessary abstractions for the underlying mechanisms. Other composition models allow explicit assembly of entities, as for example CCM.

However, our distributed AOP proposal has a reversal philosophy regarding traditional conventions of remote object or component models. Moreover, our model also allows runtime composition thanks to the use of distributed aspect descriptors. In addition, the distributed meta-pointcut provides a new way to perform distributed aspect composition in design, activation and runtime phases. Finally, current solutions usually adopt symmetric models, nevertheless we present an asymmetric model that specifies distributed aspects as the entity of the distributed middleware.

The main idea of our proof-of-concept is the fact that we aim to provide an application the ability to scale. In this way, comparing our proof-of-concept with a traditional DHT application [6], we obtain an important reduction of number of classes, lines of code, improvement of modularity, transparency, runtime changes, and, a minor complexity in general. In this setting, the DHT scenario can be considered as a p2p example in terms of application, but is totally applicable to other cases, like web server application data, the web services standard registry (Universal Description, Discovery and Integration, UDDI) [http://www.oasis-open.org/specs/index.php#uddi], or distributed databases.

The implementation of our model is publicly available on our website [http://planet.ury.es/damon], and is released under a LGPL license. The previous version was validated on the Planet-Lab network. Furthermore, we are planning further validation tests of our new middleware implementation on top of the Grid’5000 [http://www.grid5000.fr] testbed.

Furthermore, our conceptual model and contributed services are generic enough, and could be applied on some other distributed AOP frameworks, which provided similar features as Damon, but with some restrictions, like scalability or communication primitives.

Finally, a future research line is building distributed design patterns using our middleware. We think that distributed AOP is a very good area to implement distributed design patterns. As a consequence, new works are emerging, and we believe our model can be extensively used to accomplish new future achievements in this line.

Acknowledgments

This work has been partially funded by the Spanish Ministry of Science and Technology through project P2GRID TIN2007-68050-C03-03, and the European Union under the 6th Framework Program, POPEYE IST-2006-034241.

References