Neutral Programming Language Aspects with GNU/EDMA *

Uncoupled Components meet AOP

D. Martínez Oliveira
University Carlos III
Avda. de la Universidad, n X, Leganes
Madrid, Spain
dmartin@dei.inf.uc3m.es

C. Fernández
University Carlos III
Avda. de la Universidad, n X, Leganes
Madrid, Spain
camino@dei.inf.uc3m.es

J. M. Dodero
University Carlos III
Avda. de la Universidad, n X, Leganes
Madrid, Spain
dodero@dei.inf.uc3m.es

ABSTRACT
GNU/EDMA [1] SIU Extension Subsystem was developed to easily integrate different systems within the GNU/EDMA programming interface. The general method/property interception capability of SIU subsystem provides an appropriate framework for Dynamic Aspect Oriented Programming. As well, GNU/EDMA neutral programming language architecture allows to set up an AOP environment where applications and aspects can be written using different programming languages.

General Terms
Aspect-Oriented Programming

1. INTRODUCTION
A great effort has been devoted to bring Aspect-Oriented Programming (AOP) and Dynamic AOP (DAOP) to an usable stage. However, most research efforts and projects in this field have been oriented towards the Java language and platform, leaving other development environments out of the main research stream. Works on Smalltalk [3] or C++ [6] reasonably cover this issue, but the bulk of research is focused on Java.

In this paper, an AOP approach is presented, based on the GNU/EDMA [1] development environment. GNU/EDMA provides a loosely coupled object-oriented and component-based programming environment, which can be used from different programming languages in different platforms.

All the GNU/EDMA features are built against the basic services it provides, so any research on the system can be easily exploited from any of the supported platforms. From a general point of view, GNU/EDMA is against the generalized trend to integrate more and more features within the programming language. Instead, GNU/EDMA keeps these features out of the programming language itself, making them possible to be used from any platform.

The GNU/EDMA system provides additional facilities beyond AOP and a wider view of applicability of AOP in different contexts. Concretely, it provides a useful set of *Unanticipated Software Evolution* (USE) capabilities, which makes easier to bound the scope of applicability of AOP techniques, and hence, it provides an appropriate framework for evaluating the AOP impact on software adaptation and evolution.

As a side discussion, the paper also intends to state the possibilities of a system like GNU/EDMA to facilitate the testing of programming approaches in and allow for quick checks of theoretical models.

The paper is organized as follows. In section 2 a review of current DAOP techniques is presented. Section 3 states the main differences between method interception an class evolution techniques in such kind of systems. In section 4 the GNU/EDMA SIU extension system is presented and the weaving process for our system is explained. Section 5 illustrates the previous topics by means of a simple example and finally, section 6 presents some conclusions and future works.

2. REVIEW OF DAOP APPROACHES
Many works have been recently published about adding dynamic behavior to programming environments and software systems in general. In the DAOP field, three systems can be highlighted, since they cover the general idea behind most current approaches.

AspectS [3] provides dynamic aspects to the Smalltalk environment. AspectS is a powerful AOP system since it is based on a very powerful environment (i.e. Smalltalk). However, its current implementations heavily relies only on that platform. As it also happens with the following systems that will be remarked, the ideas and approaches presented therein can be applied in other environments, but the current implementations prevent their use in heterogeneous systems.

*(Produces the permission block, copyright information and page numbering). For use with ACM_PROC_ARTICLE-SP.CLS V2.6SP. Supported by ACM.

Other two important DAOP systems are specifically targeted to the Java environment, which also prevents its use from other programming languages and environments. The first is the PROSE system [5], which provides a powerful and efficient implementation of dynamic aspects for the Java platform. PROSE relies on the Java Virtual Machine debugging interface [2], and then, such implementation cannot be easily ported to other platforms. The other system that is built on top of Java is JAC [4], which provides more facilities than just dynamic aspects, as well as a set of ready-made, predefined aspects for their direct use in applications. JAC uses *bytecode adaptation* to modify Java objects at load-time in order to allow the system to manipulate them. As a major advantage of JAC, it does not present any low-level dependency on the Java platform, neither virtual machine modifications nor language extensions.

All the systems described above rely heavily on the underlying programming environment (either SmallTalk or Java bytecodes). This fact makes harder to port these systems to other platforms. GNU/EDMA approach is to move all these facilities out of the implementation environment and then completely uncouple the DAOP implementation from it.

When examining deeper DAOP solutions like the described above, two main issues arise:

- In the end, all approaches try to implement object-based features. This is a sound approach, because at the dynamic level the main software unit becomes the object and not the class —at least in Object Oriented Programming (OOP).
- DAOP features are implemented in non object-based environments, which makes the implementation harder and also makes difficult to keep object and class models consistent.

Additionally, DAOP solutions heavily rely on specific platforms. This fact makes harder for them to be integrated within heterogeneous component models like CORBA, which try to provide a programming language neutral approach. For example, it is difficult to use any Java-based DAOP within a CORBA solution when it is required to apply a Java Aspect to a C++ object. Of course, a suitable solution can be built for that purpose, but this issue has not been addressed yet by current DAOP systems.

Finally, there is a trend to use DAOP as a solution for software evolution and adaptation. From our point of view, it is necessary to set clear the scope of application of AOP, and then select the best-fit approach to each problem. In the general case, software evolution and adaptation cannot be considered as a process of adding cross-cutting or orthogonal behaviors, but it is a class/object-specific and semantically dependant process that requires special solutions. Moreover, the need to bring object-based features to class-based environments opens the possibility of better evolution and adaptation solutions with a better performance and, at the same time, maintaining the cohesion within the software that is being modified.

3. METHOD INTERCEPTION VS CLASS EVOLUTION

Before describing our GNU/EDMA-based DAOP solution, a brief discussion on when to use AOP is worth it.

Within an OOP solution, when a class is evolved or adapted, the changes that are applied are specific for that class. In this situation, the powerful cross-cutting nature behind AOP becomes degenerated, since an specific aspect is required for each class being evolved or adapted. In fact, most of evolutionary and adaptation scenarios do not cross-cut class hierarchies but single hierarchy trees, and it rarely depends only on adding/updating/removing orthogonal behaviors. Moreover, in the general case object/class evolution requires to change the interfaces and transfer the state between old instances and the new adapted/evolved ones, specially when object hot-swapping is required. In such a run-time scenario, the applicability of DAOP solutions becomes difficult and tricky. Actually, most of existing DAOP solutions do not support interface modification, neither object state transfer per se. However there exists a midway case in our current discussion. Sometimes there is the need for a certain behavior to be added to a set of related classes somehow, but not only one or several classes, nor classes that are unrelated at all. In this case it does not exist a clear way to determine which is the best suitable way for evolution, and AOP or solutions like mix-in classes seem equally suitable for the problem. In these cases, specific details of the change or limitations on the programming environment (for instance, unavailability of multiple inheritance) will determine the most appropriate solution.

Current aspect-based environments are essentially based on method/property interception, or in a more general way, they are based on the interception of *join-points.* An *aspect* is a piece of code that can be attached to any join-point and then it is able to intercept invocation of methods, access to properties, etc. and inject/remove pieces of code at those points.

Aspects are commonly used to achieve separation of concerns and to apply cross-cutting behaviors to a family of classes or components. Some classical examples in AOP are to add a logging facility to a set of classes in an already-developed application, or to add access control features in a similar way.

The availability of *dynamic* AOP solutions let researchers to envision AOP as a software evolution and adaptation solution, however it is not well suited for the general case, even when it results useful in many specific situations.

Better evolutionary solutions exist that are more consistent than aspects with respecto to long-term software maintainability. Dynamic inheritance and delegation systems, along with object state transfer support, provide a more suitable framework for evolution and adaptation than general AOP/DAOP solutions.

At the same time, there are general application *changes* that are much easier to apply using AOP/DAOP techniques. For example, distributed programming is a clear example, for which AOP techniques result perfectly suitable, since

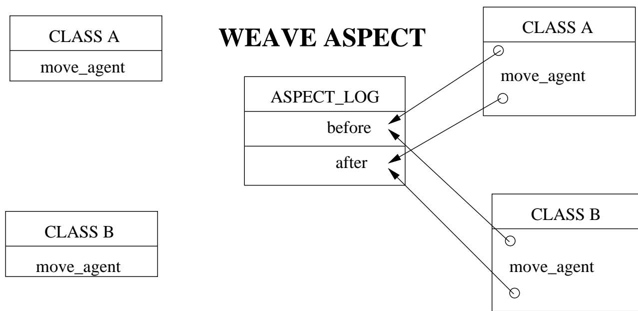

Figure 1: Debugging Log Aspect weaving

an unique general behavior (i.e. distribution) will cross-cut most of classes in the system.

To illustrate these ideas, a simple case study will be discussed. Suppose we want to add a logging service to a Mobile Agent System (MAS), like for example our AGNES MAS implementation [8], which has been developed on top of GNU/EDMA. In this situation two main scenarios may arise: a debugging log, or a server log.

3.1 Agent Debugging Log

Debugging of distributed applications that do not share a common addressing space is difficult to realize, so a debugging facility will be very useful to develop an agent-enabled application with AGNES.

In this case, it is interesting to follow the flow of execution of agents moving around a given network. Providing traces when a given method of an agent begins execution and when that method finishes will allow developers to check that the intended execution flow is respected by the system. This discussion will be centric to method execution for brevity, but it can be generalized to any other kind of join-point.

In this scenario, arbitrary methods from very different classes and objects will require logging in different moments, depending on the feature or behavior that requires debugging, that is, this logging function will cross-cut several unrelated classes/objects and, in general, it will change with time. However, such a feature is a temporary one, and is not part of any evolution process. It is just a short-time required feature, useful in a brief interval of time for a very concrete activity, which possibly will not make sense in the future.

In fact, the interesting feature of a DAOP solution is the possibility of adding an aspect, debug the application and then completely remove the aspect and the debugging functionality with it, which is no longer required, or well keep it latent until it is newly required in future.

Figure 1, illustrates this process. The `ASPECT_LOG` is weaved to different classes/objects in the systems intercepting a concrete method or set of methods and allowing to add the operations *after* and *before* these methods.

Finally, note that adding a debugging log as described so far is a completely independent action, not related at all to the semantics of the classes and objects that adopt it.

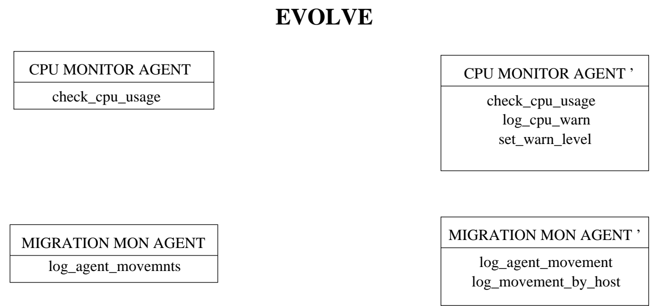

Figure 2: Each class requires a specific semantic-compatible extension

Some issues like debugging or network distribution are real cross-cutting in the sense that they are completely orthogonal to the real semantics of classes and objects using those actions/behaviors.

3.2 Agent Server Log

Now, what it is required from the mobile agents is to log information on each host they visit. In this case, since different agents are implemented as different classes, a new cross-cut scenario emerges, where several classes will share the conceptual behavior of logging, but each one will want to log different information.

This latter scenario fits better within an evolution case, where the added logging service is a missing feature which is added and, in general, will remain in the system for the future. Classes and objects really becomes other 'thing', they are not the same 'thing' with something added.

However, in this case each agent will have different logging requirements and even when the general concept of logging cross-cuts all of them, the specific details for each agent are usually different.

For example, all the agents will log their arrival to and leave from a given host, but a CPU load monitor agent will also log any alarm about excessive CPU usage, or an automatic migration agent will log which agents were chosen to be moved to other host and which host was selected for each moved agent. Figure 2 illustrates the scene. Two classes evolve in time, and in the general case, new behaviors and properties are added and removed in this process.

In this scenario, a general logging aspect (for arrival and leave of agents) is not enough, and specific aspects for each logging requirement on each agent can arise, but conceptually these are not cross-cutting features. This is a case of class evolution, which implies new versions, releases, updates or patches for existing classes.

Such a real evolution scenario can be better managed by an evolution solution where individual classes are changed. Note, however, that in the general case, when applying AOP to evolve software, a specific aspect for each one of the classes being evolved will be necessary, so better solutions can be proposed In fact, AOP solutions becomes degenerated in the sense that an specific aspect must be generated

for each specific class/object being evolved if a real evolution or adaptation functionality is required.

4. SIU EXTENSION SYSTEM

The GNU/EDMA environment, from its early versions, provides an extension system named SIU. SIU extension system was introduced to provide an easy interface to existing systems, and the main philosophy behind it is to improve code reuse even when this code belongs to a completely different environment.

The SIU extension system allows to write special GNU/EDMA classes, which are able to intercept any GNU/EDMA primitive invocation and hence, providing a mechanism to translate GNU/EDMA parameters and data to other systems, in order to execute external code within an unique programming interface. As a side effect, SIU system behaves as a general GNU/EDMA primitive interceptor, that is, it can intercept, for example, method invocations on a given object and control how its code is executed, making easy to execute code *before* or *after* the real one and even to prevent execution at all.

Using the SIU extension system, it becomes easy to build up a simple AOP system within the GNU/EDMA environment. The special SIU classes containing the interception code are called called *SIU Proxies*, and they are normal GNU/EDMA classes implementing a specific interface.

The SIU Subsystem provides two types of proxy and three levels of implementation. First type of SIU proxies are called *blind* proxies. This kind has no information about external interfaces, i.e. they know nothing about the interface of the object they represent and simply forward primitives directly without further checking, or relies on an external procedure to get any interface information that is required. Second type are *non-blind* proxies. These proxies rely on the interface information provided by GNU/EDMA in order to forward primitives. GNU/EDMA interface management is out of the scope of this paper.

Additionally, irrespective of the kind of proxies, three implementation levels are defined, which will determine the degree of integration of the proxy with the GNU/EDMA system. For our current discussion, implementation level 1 covers the classical join-point approach that is found in current AOP solutions, i.e. object creation and destruction and method and property interception. Implementation levels 2 and 3 support the interception of specific dynamic GNU/EDMA features, like dynamic inheritance and dynamic virtual method overriding. These levels are out of the scope of this paper, but they provide an interesting research direction in AOP. Level 2 and 3 proxies can become quite complex, mainly when the representative object is in an external system where these features are not supported. For the purpose of this work, Level 1 SIU proxies are used, since they cover most of the features provided by other AOP/DAOP available systems.

Finally, note that the programming language neutral nature of GNU/EDMA DAOP approach is provided as well by the SIU subsystem. Special SIU proxies are in charge of translating primitives from one system to another, at the same time that other SIU proxies implement aspects. This is the way to uncouple AOP issues from the programming environment that is actually used.

4.1 Static Aspect Weaving with GNU/EDMA

GNU/EDMA does not face static weaving as it happens in fully static systems like AspectJ [7]. This static behavior is usually addressed at the source code level at compile-time, and it is mostly independent of the underlying system. This kind of weaving does not fit well in the fully dynamic nature of the GNU/EDMA environment, but there is no technical that prevents it to be implemented.

With static aspect weaving we mean *pre-assigned weaving*, that is, the process of assigning aspects in advance to classes or objects in the system. There are two possibilities for doing this. The simplest way to associate an aspect to a GNU/EDMA class is using the GNU/EDMA registry. GNU/EDMA uses a global plain text file where classes are registered in order to allow the system to control them. Within this registry, it is easy to associate a SIU Proxy to a given class simply by editing a text line. It can be contemplated as the deployment descriptors that are used in other systems [4, 5]. First, the SIU Proxy (Aspect) must be declared, which is done by including the field `IsSIUProxy=1` to mark the class as a SIU Proxy.

```
[CLASS49]
ClassName=LOG_ASPECT
NameSpace=siu/aspects/examples
Machine=i386
OperatingSystem=LINUX
Implementation=libLOG_ASPECT.so
IDFParser=EDMAIDF
IsSIUProxy=1
MajorVer=0
MinorVer=0
CurrentVer=49
```

Then, any other class in the system can be associated with this SIU Proxy by including the field `SIUProxy=LOG_ASPECT` in its registry descriptor. Bellow there is a simple example.

```
[CLASS50]
ClassName=TARGET_CLASS
NameSpace=examples/aspects
Machine=i386
OperatingSystem=LINUX
Implementation=libTARGET_CLASS.so
IDFParser=EDMAIDF
SIUProxy=LOG_ASPECT
IsSIUProxy=1
MajorVer=0
MinorVer=0
CurrentVer=50
```

From this point on, any application in the system that creates an instance of the class `TARGET_CLASS` will associate the `LOG_ASPECT` class to that instance and get a log of any method invocation on it.

Additionally, applications can choose to associate a SIU Proxy with an object when the object is instantiated, so it is possible for applications to choose which objects of a given class will be actually logged. To perform this, the application must prefix the SIU Proxy name to the name to be instantiated, as the following example depicts.

```
OBJID target = edma_new_obj ("LOG_ASPECT:A_CLASS");
```

At the practical level, dynamic weaving is carried out. However, the possibility to weave/unweave aspects to specific classes provides also a mechanism to permanently attach/deattach aspects to specific classes in an easy way. From this point of view, this GNU/EDMA feature can be seen more like a load-time component adaptation mechanism that like an static AOP solution.

4.2 Dynamic Aspect Weaving with GNU/EDMA

GNU/EDMA also allows to associate SIU Proxies (i.e. apply aspects) to running objects, as well as to remove a previously associated proxy while the application is running. For this purpose two primitives are provided:

- `edma_attach_proxy (target, proxy)`: This primitive will attach the `proxy` (aspect) to the object `target`.
- `edma_deattach_proxy (target)`: This primitive will detach any proxy previously attached to `target`.

Following with our frame example, a fragment of code that illustrates how to use these primitives is shown below.

```
OBJID target = edma_new_obj ("TARGET_CLASS");
...
edma_attach_proxy (target, "LOG_ASPECT");
....
edma_deattach_proxy (target);
```

This code fragment creates a normal GNU/EDMA object and, at some point of its execution, attaches the `LOG_ASPECT` SIU proxy to that instance. From that point on, any method invocation on `target` will be logged to the console until when, at some point later on application execution, the proxy is detached and the logging stops.

5. A GNU/EDMA ASPECT EXAMPLE

Finally, a simple coding of the complete logging aspect that is used throughout the paper is presented in this section in order to illustrate the whole AOP process with GNU/EDMA.

As indicated above, SIU proxies are simple GNU/EDMA classes which implement an special interface. When applied to an object or class, any method invocation on that object or on any instance of that class is intercepted by the proxy, and a chance to inject code or change execution is provided.

The relevant GNU/EDMA primitives intercepted by level 1 proxies are immediately described:

- **NewObj** is executed when a new instance of a class is created. The class must have associated a SIU proxy.
- **FreeObj** is executed when an instance of a class is destroyed.
- **WProp3** is executed when a property (or member variable) of a target object is written.
- **RProp3** is executed when a property (or member variable) of a target object is read.
- **Met3** is executed when a target object method is invoked.

For our logging example, the SIU proxy only needs to implement the `Met3` method to log any method invocation on a given object. The simplest implementation for this method is shown bellow:

```
ESint32 EDMAPROC
LOG_ASPECTMet3 (OBJID proxy, OBJID target,
                EPChar method, EPVoid args)
{
  /* BEFORE code */
  edma_printf ("Method %s is about to be executed "
               "on object %d", method, target);

  edma_met3_pargs (target, method, args);

  /* AFTER code*/
  edma_printf ("Method %s object %d finish",
               method, target);
}
```

This code fragment logs a message each time that a program enters any method on the target object and each time that it is left. From the point of view of the GNU/EDMA programming environment, this aspect is a normal GNU/EDMA class. Now, the aspect can be dynamically weaved to any class or object in the system using the different possibilities described in previous sections.

6. CONCLUSIONS AND FUTURE WORKS

This paper introduces the GNU/EDMA SIU extension subsystem and its application to AOP/DAOP. This GNU/EDMA based approach provides the following advantages when compared with current available similar systems:

- Aspects can be applied to different programming environments and can be written using different programming environment.
- It homogeneously integrates within the GNU/EDMA programming environment. No additional aspect-based language must be learned and there are no differences with respect to normal development of GNU/EDMA classes.
- It provides a dynamic AOP solution that complies with the "all-dynamic" philosophy behind the GNU/EDMA system.

Up to the knowledge of the authors, GNU/EDMA SIU Proxies are the first approach to face language and programming environment neutral approach to AOP. Although there are some approaches to support different programming languages, as the proposed in [9], which uses the .Net framework to achieve this aim, GNU/EDMA approach is more general than a simple programming language support, since it can also be used in different programming environments, like component-based systems or distributed systems.

Since SIU Proxies were not developed with AOP in mind, additional work is required to deal with common AOP issues that are already addressed by other platforms. From this point, the results obtained by other research efforts in the DAOP field will be a valuable help to complete a DAOP platform on top of GNU/EDMA.

At present, GNU/EDMA provides partial support for Perl, Python and C# programming languages throughout SIU proxies, as well as the C/C++ native interface. Java integration is also planned. These proxies are ready to use these programming languages as implementation languages for GNU/EDMA classes, and then use them from the main application programming interface. At the same time, for each programming language it is required to write a simple interface module to allow the access to GNU/EDMA primitives.

A key point at this moment is to clearly define where AOP/DAOP and other specific software evolution techniques will be more useful, in order to clearly delimit the scope of applicability for each one of them. The incorporation of DAOP support to the GNU/EDMA system has allowed to use evolutionary techniques and aspect-oriented solutions within a unique environment, thus making easier to bound the applicability of these techniques.

SIU Proxies of level 2 and 3 may also provide interesting results for AOP that have not yet been explored. At these levels, aspects can be weaved when super or subclasses are attached or detached to a given object and when virtual methods are overridden. Since dynamic inheritance is not a common feature on current systems, this is still an unexplored application field for aspects.

Finally, the GNU/EDMA programming environment also provides a very powerful MOP (Meta-Object Protocol) tool, which provides with a quite simple interface for quick testing of new ideas in the general OOP world and, in particular, the AOP field.

7. ADDITIONAL AUTHORS

No additional authors

8. REFERENCES

[1] David Martinez *GNU/EDMA Web Page* http://www.gnu.org/software/edma

[2] Sun Microsystems *JavaTM Virtual Machine Debug Interface Reference* Available on-line: http://java.sun.com/j2se/1.3/docs/guide/jpda/jvmdi-spec.html

[3] Robert Hirschfeld *AspectS ?- Aspect-Oriented Programming with Squeak* Objects, Components, Architectures, Services, and Applications for a Networked World, pp. 216-232, LNCS 2591, Springer, 2003

[4] R. Pawlak, L. Senturier, L. Duchien, G. Florin *JAC: A flexible solution for aspect-oriented programming in Java* Reflection 2001, Koyota, Japan 2001

[5] A. Popovic, T. Gross, G. Alonso *Dynamic weaving for aspect oriented programming* 1st Intl. Conf. On Aspect-Oriented Software Development, Enshede, The Netherlands, 2002

[6] O. Spinczyk, A. Gal, W. Shröder-Preikschat *Aspcet C++: An aspect-oriented extension to the C++ Programming Languages* Fortieth Int. Conf. on Technology of Object-Oriented Languages and Systems, TOOLS, Pacific, 2002

[7] Palo Alto Research Center *http://aspectj.org*

[8] David Martinez *AGNES: GNU/EDMA Mobile Agents* http://www.dei.inf.uc3m.es/ dmartin/agnes

[9] Wolfgang Schult, Andreas Polze *Dynamic Aspect-Weaving with .NET* Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen, TU Berlin, Germany, 7-8 November 2002