Proceedings of the 2004 Dynamic Aspects Workshop (DAW04)

Robert E. Filman
Michael Haupt
Katharina Mehner
Mira Mezini

RIACS Technical Report 04.01

March 2004

Lancaster, England
March 2004

Proceedings of the 2004 Dynamic Aspects Workshop (DAW04)

Robert E. Filman, RIACS
Michael Haupt, Darmstadt University of Technology
Katharina Mehner, Technical University Berlin
Mira Mezini, Darmstadt University of Technology

RIACS Technical Report 04.01

March 2004

Lancaster, England
March 2004

This volume represents the proceedings of the Dynamic Aspects Workshop, held at the Aspect-Oriented Software Development conference in Lancaster, England in March, 2004. This workshop identifies examples of useful dynamic aspect behavior, suggests appropriate linguistic structures for dynamic aspects, and discusses implementation techniques for dynamic aspects, such as shadow compilation and modifications required in the underlying execution environment.

Table of Contents

Workshop Organization

Workshop Organizers

- **Robert Filman**
  Research Institute for Advanced Computer Science/NASA Ames
- **Michael Haupt**
  Darmstadt University of Technology
- **Katharina Mehner**
  Technical University Berlin
- **Mira Mezini**
  Darmstadt University of Technology

Program Committee

- **Gustavo Alonso**
  Swiss Federal Institute of Technology (ETH Zürich)
- **Kris de Volder**
  University of British Columbia
- **Robert Filman**
  RIACS/NASA Ames Research Center
- **Michael Haupt**
  Darmstadt University of Technology
- **Klaus Havelund**
  Kestrel Technologies / NASA Ames Research Center
- **Stephan Herrmann**
  Technical University Berlin
- **Robert Hirschfeld**
  DoCoMo Euro-Labs
- **Guenter Kniesel**
  Universities of Osnabrück and of Bonn
- **Crista Lopes**
  University of California, Irvine
- **Katharina Mehner**
  Technical University Berlin
- **Mira Mezini**
  Darmstadt University of Technology
- **Christa Schwanninger**
  Siemens CT/SE

Introduction

Join points are the locus of aspect and functional code interaction. Traditional aspect systems define join points in terms of the static structure of programs. Method wrappers, before and after method calls, variable setting and retrieving, and structural identification of object fields are all examples of traditional join points.

Beginning with the "Jumping Aspects" paper of Brichau et al., examples began to emerge in which it was desirable to invoke or change aspect behavior based on the dynamics of program execution. Such situations include changing behavior based on the call-stack context, co-occurrence of predicate triggers, concurrent thread status, or events in the underlying interpreter such as storage reclamation or process scheduling. Cflow in AspectJ was one response to the need for dynamic aspects.

This workshop identifies examples of useful dynamic aspect behavior, suggests appropriate linguistic structures for dynamic aspects, and discusses implementation techniques for dynamic aspects, such as shadow compilation and modifications required in the underlying execution environment. The topics of the workshop include:

- Applications of dynamic aspects,
- Models for dynamic aspects,
- Techniques for validating dynamic aspect programs,
- Linguistic structures for dynamic aspects,
- Dynamic action expression and linguistic integration,
- Implementation mechanisms for dynamic aspects,
- Enabling technologies and environment support for dynamic aspects (e.g., debuggers, IDEs),
- The relationship between dynamic aspects and the rest of a software system,
- Challenges and research directions.

A Dynamic Aspect Oriented C++ using MOP with Minimal Hook Weaving Approach

Sufyan Almajali and Tzilla Elrad
Department of Computer Science
Illinois Institute of Technology
{almasuf,elrad}@iit.edu

Abstract

This paper presents a dynamic aspect oriented system using C++ programming language The work uses the concept of Metaobject Protocol (MOP) along with Minimal Hook weaving approach to support dynamic weaving and unweaving of aspects at run-time. The aim of this work is to present the first Dynamic AOP system running on C++. Also, this work focuses on the suitability of Dynamic Aspect Oriented C++ (DAO C++) for networking systems.

1 Introduction

Crosscutting concerns are software concerns that under a given decomposition of a system into modules can not be implemented in one place. Examples of crosscutting concerns for networking include security protocols, fault tolerance protocols, and quality of service. Aspect-Oriented Programming (AOP) [1,4] is an emerging technology for modularizing crosscutting concerns. We are applying AOP technology to the design and implementation of networks. Networking technology is characterized by having to get many pieces of software to work together seamlessly. AOP provides the ability to insert new behaviors into software systems, and to make these changes in a coordinated manner across a software system. Many of the networking systems require efficient run-time adaptation of services, protocols and sometimes even system code. For example, in networking mobile systems, the behavior of a mobile node can change based on its environment, the encryption protocol can change as nodes move, new security code patches and hot fixes may need to be loaded and integrated at run-time, and auditing and logging features may need to be turned on and off.

Run-time adaptations that reduce availability lose revenue and customers [5]. Because of the need for continuous operation, planned downtime is hard to schedule, and unplanned system halts can cause large losses for these network providers and organizations. When a new service or security threat makes changes unavoidable, the changes applied might be scattered in many places. This reduces system modularity and increases coupling. At the same time, ignoring needed maintenance and development causes the system to become outdated, eventually to the point of becoming an obstacle to organizational development [5].

Recently, there has been a growing interest in using dynamic aspect-oriented techniques. These techniques allow run time weaving and unweaving of aspects. Examples of these approaches: PROSE system [9], JIT [10], HandiWrap [10], JAC [8], and others. We compare these systems in Section 6; for the moment it is important to understand that these are all Java systems, and achieve dynamic, aspect-related changes by taking advantage of properties of the Java Virtual Machine. Java is insufficiently efficient for networking. In this paper, we describe the DAO C++ system, which realizes dynamic aspects in C++. Features of DAO C++ include:

- **Dynamic Weaving.** Aspects can be woven and unwoven at run time using C++.
- **Efficient execution.** Programs execute efficiently under normal operation.

The paper is structured as follows: In Section 2 we describe several requirements on networking systems and how to achieve them using dynamic AOP. In Section 3 we define the

basic architecture of our proposed system. The implementation details of our system are explained in Section 4. The system performance is examined in Section 5. A comparison to related work is presented in Section 6. We conclude the paper with section 7.

2 Motivations and Goals

2.1 Dynamic Adaptable Design

One of the goals of this work is to produce dynamic adaptable systems. This allows network systems to adapt to new protocols, services and functionality at run-time without stopping the system. In networking terms, this means making the networking system an adaptive network node (ANN) [12]. ANNs reduces system downtime and increases the productivity of networking systems. To achieve this goal, we need two features:

- Dynamic weaving and unweaving of Aspects.
- Run-time code modification.

2.2 Flexibility and Modularity of Networking Systems

Networking organizations sometimes develop new standards including both brand new standards and new versions of old standards. The proposed AOP technique facilitates transitioning to new network architectures and protocols. For instance, assume that we want to evolve to IPv9, as a successor to IPv4/v6. In an Aspect-Oriented system such as ours, this requires changing only the protocol definition module. In a conventional system, we would have to find all places in every module where the assumptions and implementation of IP v4/v6 have been realized, and modify those locations. This same principle applies to other concerns of the system. We can, for example, change the queuing model without modifying the IPv4/v6 code, despite the fact that the instructions of these two modules compile together.

2.3 Efficiency

For many networking systems, such as switches, routers, and mobile nodes, performance is critical. Current dynamic AOP techniques run on top of the JVM, which is too slow for serious networking, especially in comparison to “closer-to-the-machine” languages like C++ [5]. This work presents first dynamic AOP system running on C++.

2.4 Rapid Prototyping and Debugging

Different system components, modules, protocols and services can be tested at run-time without the need to recompile, reinstall or restart the system. This will speed up system evolution and testing process [9]. System engineers and programmers can test different system behaviors in a shorter period of time.

2.5 One Language Setup Approach for Dynamic AOP

Our work makes the source language the aspect language. Using one language setup approach makes it easier on programmers to apply the concepts of AOP in their systems. The general purpose C++ programming language will be used to write dynamic AOP systems. Programmers do not need to learn a new language.

3 The Basic Architecture

Our DAO C++ architecture works using two components: preprocessor and AOP engine (Figure 1). The preprocessor works pre the compilation stage and is used to generate the necessary metaobject data that will be needed at run-time. Moreover, the preprocessor modifies the original source code and insert the minimal hooks required to support dynamic weaving. In addition, the modification includes adding generated metaobject data to become part of the new source code. After this, the new source code will be passed to C++ compiler. After compilation and loading, running program will contain metaobject data about program classes and methods. This information will be used to achieve dynamic aspect weaving and unweaving by the AOP engine.

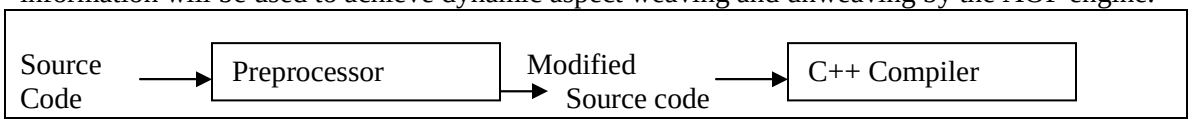

Figure 1. Preprocessing Stage of DAO C++

The AOP engine works at run-time and it accomplishes the following tasks:

- Run time Join-Point generation.
- Aspect Weaving and Unweaving.
- Aspect Advice Execution.

At run-time, user sends a request to the AOP engine in order to weave a specific user defined aspect. AOP engine weaves aspects by doing the following steps (Figure 2):

1. Read Aspect matching expression: this will be a user-defined expression.
2. Search through metaobject data to find the matching classes/methods that match the aspect expression.
3. For all classes/methods that match the given aspect expression, minimal hooks will be activated and linked to given aspect.
4. Aspect will be registered as woven aspect.

After that, when the program sequence of control reaches a specific minimal hook, control is passed form program to the AOP engine. Then, the AOP engine runs the appropriate advice(s) for whatever aspect(s) activated on that join-point (activated minimal hook).

A user can send a request to unweave a specific aspect. The AOP engine receives the user request and deactivates the corresponding hooks for that specific aspect.

Delaying aspect join-points generation to run-time adds extra overhead during weaving time. However, this design decision has been made on purpose to support extreme flexibility by allowing aspect scope to change at run time. This nice feature of aspect reshaping at run-time allows user to minimize number of aspects needed in addition to the number of weaving and unweaving steps. AOP engine supports refresh method that can be called explicitly by an application programmer to reflect latest aspect matching expression. Other approaches can be followed for join-point generation like doing it at compilation time. This will lead to two system drawbacks: less flexible aspect and extra space requirement. Nevertheless, it can improve system performance during weaving and unweaving time.

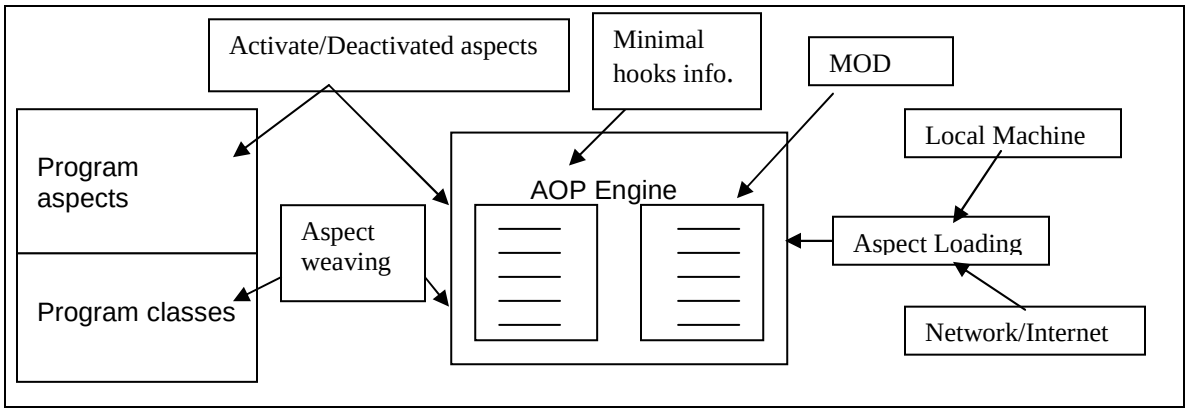

Figure 2. AOP Engine at Run Time

4 Implementation

4.1 Aspect and AOP Engine Classes Implementation

Before we get into details, let us take example that shows us the syntax of defining aspects and weaving/unweaving/refreshing them using our DAO C++ system. To include the DAO C++ capability into a normal C++ program, two classes should be used:

- The "Aspect" class, and
- The "AOP_Engine" class.

Application programmers to define their user-defined aspects use the aspect class. User extends the aspect class by inheriting from Aspect class and overrides the definition of advice method. Moreover, user defines aspect join-points by defining aspect expression. This can be done at aspect definition time, or it can be done after aspect instantiation. Figure 3.a illustrates the idea of aspect definition.

The aspect expression has two main parts: class and method. The class part is used to narrow the scope of aspect join-points whenever we want to limit the weaving to a specific class. Using the wildcard character "%" can ignore class part. The method part defines the method signature that includes return type, method's name, and method's parameters.

The special wildcard character can be used to replace any of the method signature's parts. Additionally, a complex expression can be formed using the "or" operation represented by "||". The "or" symbol will be part of the string and will be analyzed by the AOP engine during join-point creating time. Analogously, the "and" operation "&&" can be used. Besides the aspect matching expression, two Boolean attributes are used to locate the type of advice execution: before method execution, after method execution or both. This structure of aspect defined as class, allows an application programmer to define their own attributes and methods inside the aspect where these are accessed only locally from within the advice code. Defining a new method that accesses classes outside the aspect class scope can lead to indirect recursion and deadlock cases.

The second class that should be used by an application programmer is The "AOP_Engine**"** class. This class is already defined in a special C++ header file that has all of the related DAO C++ classes and Metaobject data related data structures. This file is named "daocpp.h".

```
class NewAspect : public  Aspect {
   void  advice( )
   {
       advice code
    }
   } // end of aspect
In the main program:
NewAspect  na;
na.setExpression("class
class_name_match,method
method_signature()  ")
na.setBefor(true);
na.setAfter(flase);
```

```
AOP_Engine   aop_engine;
.
aop_engine.weave(&na);

// note: na is user defined aspect.

aop_engine.unweave(&na);

aop_engine.refresh(&na);
```

Figure 3. (a) Aspect Definition in DAO C++ (b) AOP_Engine Definition

In order for an application programmer to weave a specific aspect, he should create an instance of the AOP_Engine first. Then he invokes the weave method on a specific aspect. Figure 3.b illustrates how this works. Similarly, user can invoke unweave and refresh methods.

4.2 Preprocessor Implementation Details

The preprocessor works before the compilation stage and is used to achieve two tasks:

- Insert minimal hooks in appropriate locations
- Generate necessary Metaobject data that will be needed at run-time.

To achieve this, preprocessor program scans the source code and does the following:

- It creates a class record for each parsed class. This class record holds information such as the class name, number of methods, and pointers to method records.
- It creates a record for each method definition that holds information such as the method name, number of parameters, types of parameter and return type.
- It inserts a minimal hook at the beginning of each method body, as shown in Figure 4.
- It inserts a minimal hook at each possible method exit point.

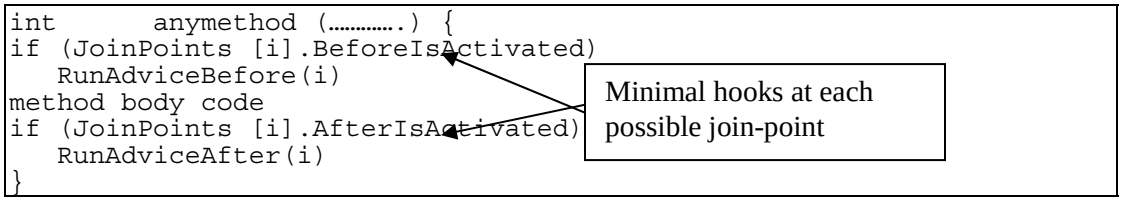

Figure 4. Minimal Hook Insertion by DAO C++Preprocessor

We have implemented our preprocessor component using OpenC++ metaobject protocol [3]. OpenC++ is a useful toolkit to develop C++ source-to-source translators and source code analyzers.

At the end of preprocessing stage, Metaobject data will be ready for instantiation during program loading time. Metaobject data includes two important objects join-points and ClassInfo. JoinPoints is a list of all possible join-points (hooks). Each inserted hook is linked to one of these join-points. The Classinfo object includes all necessary information for join-points generation at run-time. Figure 5 illustrates the idea of using Classinfo and join-points.

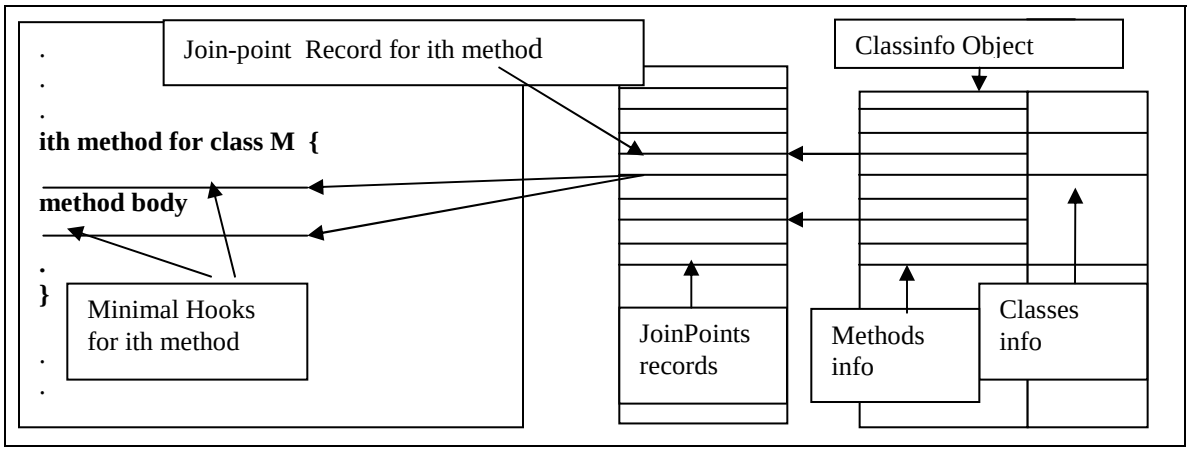

Figure 5. Classinfo and join-points objects generated by DAO C++ preprocessor

4.3 Run-Time AOP Engine Implementation Details

During program running, all metaobject data will be available to our AOP engine. When a user requests a specific aspect weaving, the AOP engine reads the aspect matching expression and starts scanning its Classinfo object searching for matches. Each time a match is found, the corresponding join-point record will be modified to reflect join-point activation. Woven aspect will be registered at that join-point. Because minimal hooks are linked to join-points, this will reflect the aspect weaving into the program.

When the program sequence of execution reaches a specific minimal hook, control is passed from program to the AOP engine. Then, the AOP engine runs the appropriate advice(s) for whatever aspect(s) activated on that join-point. AOP engine utilizes the virtual function feature to call the advice function for each woven aspect.

4.4 Synchronization of Weaving and Unweaving Processes

In many cases, user may request to unweave a running aspect or weave a new one. This can lead to inconsistency problems by having part of the join-points disabled and other part active for a specific new or old aspect. To avoid this problem, a semaphore solution has been used. A semaphore variable has been added to each aspect. In case of weaving a new aspect, the AOP engine will have mutual exclusion access to the aspect until it is done with weaving that aspect. Program classes will not be able to access the aspect unless semaphore has been released by AOP engine. This guarantees atomic weaving of aspect at all join-points.

In case of unweaving a running aspect, this is more complicated case. The AOP engine requests mutual exclusion access to the aspect. If the aspect advice is getting executed, then AOP engine waits until all classes running that aspect exit the advice code. No new access will be granted to run aspect advice code at that time. Once all current classes running the aspect code are done, AOP engine is granted access. Until AOP engine is done with the unweaving process no one can access the aspect code. After AOP engine is done, if there was any pending request(s) from classes to access the aspect code during or before weaving process, a default action will be executed. Default AOP engine action does nothing.

This simple solution has its drawbacks and a lot of work needs to be done to improve it. In case if we have a class depends on another and both use same aspect, a deadlock status can take place.

5 Performance Analysis

To measure the time complexity of our DAO C++ system, we ran several tests. One test was to estimate the time needed to execute a statically woven program. Another test was to estimate the time of executing an aspect free program in our DAO C++. The last test was to measure the performance while aspects are woven.

DAO C++ has added 7% extra overhead in executing normal C++ program that has no aspects woven. This extra overhead is due to the minimal hooks checks that check if corresponding joint is enable or not.

A more relevant comparison is between static aspect weaving and dynamic aspect. A manual static weaving has been done and compared to the dynamic aspect weaving using our DAO C++. Around 37% performance overhead has been added by our system in this case. This high overhead is due to the advice execution redirection from join-points to AOP engine. DAO C++ is still in its first version and extra work should be done to reduce this performance overhead.

6 Related Work

The idea of supporting dynamic Aspect weaving and unweaving is not new in AOP. Many techniques have been proposed and implemented. PROSE is one of the main ones [9]. PROSE is implemented on JVM and relies on built-in java reflection capabilities. The one language setup approach used by PROSE has been followed by our DAO C++ work. Just in Time (JIT) aspects allows dynamic weaving and unweaving more efficiently [10]. JIT is implemented in Java too.

Handi-Wrap is implemented in Maya as an extension to the Maya programming language. Maya is a compile-time metaprogramming system that allows users to define language extensions [2]. JAC allows Java methods to be dynamically wrapped and unwrapped through a classloader that performs bytecode rewriting [8]. JAC allows aspect to be unwrapped while Handi-Wrap allows only wrapping. It is interesting to know that JIT, Handi-Wrap and JAC use minimal hook weaving approach to support dynamic weaving.

All of the techniques mentioned above are implemented using Java. Java performance makes it unsuitable for specific type of applications.

AspectC++ supports AOP in C++[11]. Weaving is done at the compilation time like in AspectJ. This is known as static weaving. In static weaving, the concept of aspect does not exist at running time. So AspectC++ can not support aspect weaving at run time, a desired feature for some applications. Static weaving adds no overhead at the run-time. As in AspectJ [6], AspectC++ provides a rich set of joint point definitions and advice types that are not offered by our model for now.

In the MOP area, many metaobject protocols have been designed. OpenC++ [3] and CLOS [7] are two common ones. In general, MOPs is used to reveal program information for debugging and performance analysis concerns. It allows programmers to view program metaobject data at compilation or run-time, but in general it does not allow program modification. Also, space complexity for these protocols is high. As a consequence, we decided to design our own simple MOP.

Techniques for dynamically changing a running C++ program already exist. Dynamic C++ Classes allow run-time updates of an executing C++ program at the class level [5]. Also, many dynamic linkers have been introduced for C and C++ [5]. All of these techniques work at the class or function level. They do not recognize the concept of aspect.

7 Conclusion and Future Work

This paper describes the first dynamic AOP technique for C++. DAO C++ supports dynamic weaving and unweaving of aspects at run-time. Our work uses the concept of Metaobject Protocol (MOP) along with Minimal Hook weaving approach to support dynamic AOP. The main goal of

DAO C++ is to provide suitable environment for networking systems that need efficient and adaptable implementation. DAO C++ provides reasonable efficiency under normal execution after aspect is woven. DAO C++ is still in its early stage. Integrating dynamic code modification with DAO C++ will make DAO C++ powerful tool for many networking systems.

References

[1] M. Akşit, S. Clarke, T. Elrad, and R. Filman. *Aspect Oriented Software Development*. To be published by Addison-Wesley, Reading Mass., 2004.

[2] J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In *1st Intl. Conf. on Aspect-Oriented Software Development*, Enschede, The Netherlands, pages 86-95, Apr. 2002.

[3] S. Chiba. A Metaobject Protocol for C++. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA*), page 285-299, October 1995.

[4] T. Elrad, R. Filman, and A. Bader. Aspect-oriented programming. *Comm. ACM*, 44(10) 2001, 29–32.

[5] G. Hjálmtýsson, and R. Gray. Dynamic C++ Classes A lightweight mechanism to update code in a running program , in *Proc. USENIX Annual Technical Conference*, pp. 65-76, June, 1998

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, An overview of AspectJ. *Proc. ECOOP 2001*, LNCS 2072 (Berlin, June 2001), J. L. Knudsen, Ed., Springer-Verlag, 327–353.

[7] G. Kizales, J. des Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*. The MIT press, 1991.

[8] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible solution for aspect-oriented programming in Java. In *Reflection 2001*, pages 1–24, Koyota, Japan 2001.

[9] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In *1st Intl. Conf. On Aspect-Oriented Software Development*, Enschede, The Netherlands, Apr. 2002.

[10] A. Popovici, G. Alonso, and T. Gross. Just-in-Time Aspects: Efficient Dynamic Weaving for Java. In Proc. *2nd International Conference on Aspect-Oriented Software Development*. Boston, Pages: 100 – 109, 2003

[11] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10 of Conferences in Research and Practice in Information Technology. ACS, 2002.

[12] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine* 35(1), 1997, 80–86.

Hot-Deployment of Aspects

Rainer Burgstaller
Thomas Fritz
Egon Wuchner

Siemens AG
Otto-Hahn-Ring 6
81739 Munich
Germany

rainer.burgstaller@fh-hagenberg.at
fritzt@informatik.uni-muenchen.de
egon.wuchner@siemens.com

Abstract

Dynamic Aspect Weaving is a powerful mechanism that enables the weaving of aspects into an application at run-time. Most current Aspect-Oriented Programming approaches do not fully exploit the dynamic approach of weaving. They require the aspect code to be written before the compilation of the application they are woven into. We propose an approach to dynamic aspect weaving that offers hot-deployment of aspects, i.e. aspects can be enabled/disabled and added/removed at run-time. In the following we will explain our approach, look at it from a more theoretical point of view by introducing the notion of contextual polymorphism and motivate the necessity and benefit of using dynamic aspect weaving in a more exhaustive way. We complete by describing the specifics of our prototypical implementation on .NET.

1 Introduction

Aspect-oriented programming supports the introduction of crosscutting concerns in a modular way. This can be done at compile-time if the crosscutting concerns are known at this early point of time. But sometimes it is advantageous to induce or change crosscutting concerns at run-time without having to know them or their specialties in advance. In the following we will motivate our approach with two examples, one from the steel manufacturing and the other one from the telecommunication domain.

Imagine a company that delivers whole or parts of rolling mills together with a lot of controlling software to steel manufacturing companies. Every rolling mill has very special requirements and thus every time a new plant is built, a lot of customization and tuning of the software is necessary. Therefore in most cases a lot of data is written into trace files and databases when the plant is running to be able to do the right adaptations. Even though only few tracing messages, e.g. special data about the temperature or the pressure, are needed and this just for the customization, all possible data has to be traced all time. It is not possible to know in advance the exact data that is required for the special fine-tuning of the machine. The programmers then have to find the important tracing messages out of the big bulk of data, to calibrate the plant correctly.

The same happens when the plant has to be fine-tuned for a special production procedure during runtime, without stopping the execution. Therefore again the programmers have to filter

the necessary data out of the big amount of traced messages. For this case, the data either has to be traced all the time or the software has to be updated on-the-fly with new tracing functionality. But the latter one is very cumbersome and error prone as the machine has to run 24 hours a day and must not be stopped for software updates.

What we need is an approach to easily write several special tracing aspects, e.g. one or more for the temperature, enable the ones needed for the fine-tuning and disable them afterwards. Furthermore it should provide the possibility of adding and removing aspects at runtime, so that in case new tracing functionality is needed, you can just add new aspects while the application is running, without the need of a software-update.

Therefore we want to introduce an environment that uses dynamic aspect weaving for adding, removing, enabling and disabling of aspects while the application is running.

Another example is an ISDN-LAN-Gateway, which transforms telephone calls from ISDN to a telephone protocol based on IP. Such a gateway handles up to 20 call sessions in parallel at any time a day and it has to run 24 hours a day since communication must be assured all the time. If a user reports a lapse of the gateway, product support people have to trace/test the parallel course of events of the call traffic. Therefore the gateway code must be instrumented with tracing statements to record the time stamp messages and then the instrumented code is run on a test-gateway in the lab. With the information reported by the user, calls are simulated to reproduce the behavior and thereby the support people try to find the code causing the lapse. But in the lab it is very hard and time expensive to reproduce the gateway in the field. With our new approach you could easily instrument the running gateway in the field by adding and enabling a tracing aspect and see when it fails for the user without the need of simulation in a lab.

The remainder of this paper is organized as follows. In Section 2 we are going to explain our comprehension of hot-deployment of aspects in combination with dynamic aspect weaving and relate it to existent approaches. This is followed by the introduction of contextual polymorphism as a theoretical foundation in Section 3. This section also provides a short comparison of contextual polymorphism to related approaches. Section 4 gives an overview of the realization in .NET. Finally, Section 5 summarizes the paper.

2 Hot-Deployment of Aspects

Dynamic aspect weaving is a mechanism that allows aspect code to be woven into an application at any point of execution. Our approach is based on this concept to provide hot-deployment of aspects. That means aspects can be

- added and removed at runtime, i.e. completely new aspect code is attached to the system in use; e.g. the new tracing aspect for the ISDN-LAN-Gateway that was written for a special error reported by the user and added to the application while it was running;
- enabled and disabled at runtime, i.e. aspects already existent in the code of the running application can be activated and deactivated; e.g. the tracing aspects that have to be enabled only for the fine-tuning of special procedures.

Thus new aspect-oriented behavior can be woven in and out of running applications and can be activated and deactivated.

2.1 Related Approaches

Probably the most popular application for aspect weaving is AspectJ [1]. In AspectJ the aspects are woven into the (source or byte) code at compile time requiring the aspects to be written beforehand. Thus the aspect-oriented behavior is added statically to the application. Only the evaluation of pointcuts that refer to the control flow (cflow) is done dynamically.

JManlger [2] and JMunger [3] can adapt classes at load-time without requiring the source code. Both approaches work on the lowest level of dynamic weaving, i.e. you can add aspects at load-time but have no possibility of removing them later on. This approach works with a modified version of the default Java class loader. The class loader uses transformers for changing the functionality of existing classes. A transformer (JMangler's counterpart to the traditional Aspect Weaver) can transform classes through dynamic checks at all potential join points. These checks decide whether aspects should be applied to the join points or not. As JMunger is running on top of JMangler, it works in exactly the same way. The only difference is that it provides some pre-defined transformers for easier use. These two approaches can activate aspect code during runtime, but the activation is not reversible. Furthermore transformers have to be written statically before the application, to which they should be applied, starts. So there is currently no possibility to add, remove or disable aspects once the application is running.

With Schult and Polze's approach [4] for .NET it is possible to decide based on late binding at instantiation-time, whether an object should be mingled with an aspect or not. By using this kind of dynamic aspect weaving, it is possible to enable aspects at runtime. But the aspect-oriented code with the condition that specifies when to weave the aspect has to be in the source code before the compilation of the program. One major drawback of the approach is the irreversibility of interwoven aspects. Once an aspect is woven in, there is no way to "unweave" it again at runtime.

As in the aforementioned approaches, in Caesar [9], a language introducing aspectual collaboration interfaces, the aspect-oriented code has to be written and added statically before compiling the program. However the weaving is done by an extended Virtual Machine (Steamloom [5], see 3 Contextual Polymorphism for a description) at runtime. Caesar provides a mechanism for dynamic deployment and un-deployment of aspects, the so-called deploy block. The aspects to be woven in are determined at runtime and will only affect the code in the deploy block. Thus the approach offers a mean to enable and disable aspects at runtime where the aspect code has to be written before compile-time.

JBoss AOP [6] is a framework that provides a hot-deploy feature for weaving in new aspects at runtime. To use this feature, all classes that will be affected by aspects have to be instrumented, i.e. all possible join points in the class are decorated with hook methods. Once the application is running, the possible set of join points can not be extended or reduced. To be sure that any aspect requiring an arbitrary set of join points can be added later, all join points would have to be decorated with hook methods before runtime. This leads to a large performance overhead. JAC [7] is very similar to the JBoss approach in the way that new aspects can be added at runtime and that join points have to be aware of aspects that might be added later on. Hence it leads to the same performance overhead as JBoss AOP.

So the approaches mentioned above either accept a large performance overhead (JBoss AOP, JAC) to provide a hot-deploy feature in our sense of dynamic aspect weaving, or they do just provide means for enabling and disabling aspects at runtime, like the deploy block in Caesar/Steamloom.

3 Contextual Polymorphism

Before discussing the relationship of hot-deployment and dynamic weaving in more detail we are going to elaborate on the features of hot-deployment (enabling/disabling and adding/removing aspects) and their conceptual implications. In the following we keep to the terminology of AspectJ.

Aspect code like advice in the sense of AspectJ is often linked to the original code by specifying the points in the control flow of the original code where the aspect code should be executed during run-time. Essentially, class behavior captured by a certain method varies due to the set of aspects affecting this method. This is most obvious with respect to "execution" join-

points but it is not only restricted to them. For example a “call” join-point extends the implementation of the caller method instead of changing the callee behavior like an execution join-point. In a broader sense the same holds true for other join-points (e.g. “get”/”set”) and combinations of join-points as well (e.g. “cflow” combined with one of the previously mentioned join-points). Consequently, enabling/disabling and adding/removing aspects change the aspectual context of an affected method.

This has some similarities to the mechanism of object-oriented polymorphism. Overriding a virtual method represents one possible dimension of variation within a class hierarchy. The set of aspects possibly valid for a method can be regarded as another dimension of method overriding. Before, after, around advice and any precedence declarations among the set of aspects can be aligned along this dimension. Additionally, any proceed statement within an around advice of such an aspect (in the sense of AspectJ) can be compared to a virtual method calling its base class equivalent. Proceed and base class calls, both operate within their dimension towards the original method code.

As in case of object-oriented polymorphism the exact method variation to be executed can only be resolved at run-time. The actual set of aspects and their advice code to be applied to a method depends on its run-time context of enabled/disabled and added/removed aspects. This valid set of aspects specific to a method at run-time is a subset of the statically determinable set of enabled aspects at startup and the aspects being added in the meantime with potential relevance to this method. Therefore we denote this mechanism of method overriding *contextual polymorphism* analogous to the well-known object-oriented way of polymorphism.

Thus contextual polymorphism covers both, the enabling (disabling) of aspects and the addition (removal) of extra aspects at run-time. The approach of dynamic weaving undertaken by Steamloom [5] harnesses the internal interpreter handling of object-oriented polymorphism in order to implement its contextual counterpart without explicitly introducing the notion of contextual polymorphism.

Steamloom uses the open source JVM implementation of IBM called Jikes. It allows to access the virtual method table of a class by accessing the so-called type information block of an object. The explicit method body the specific advice code should be applied to is replaced with code containing advice invocations and the original code. Considering the previous elaboration on the similarities of object-oriented and contextual polymorphism this seems to be a natural idea for dynamic aspect weaving.

3.1 Difference of Object-Oriented and Contextual Polymorphism

Object-oriented polymorphism mainly works by replacing the virtual method implementation. Compared to this substitutive character of virtual method overriding contextual polymorphism is basically an additive, though reversible way of method variation. Statically, this is mainly supported by the use of proceed calls within around advice. Calling proceed is rarely omitted within an around advice. On the other hand dynamic enabling/disabling and adding/removing aspects mainly contributes to this reversible additiveness as well.

Because of the additive nature of contextual polymorphism the side effect of this kind of variation is much more extensive than the dispatch of another implementation of a virtual method. For example enabling a larger set of aspects of a method is a major behavioral change to it. Therefore it seems appropriate to minimize the impact of contextual polymorphism to the current thread of execution, thus implying a thread-specific enabling and disabling of aspects.

3.2 Contextual and Aspect(ual) Polymorphism

Previous work ([8], [9], [10]) has already done some investigation on the concept of polymorphism with respect to aspects and aspect-oriented programming. We are going to

introduce these studies in order to describe the similarities and differences to contextual polymorphism.

Ernst and Lorenz [8] take a closer look at AspectJ and its support of object-oriented polymorphism within a hierarchy of aspects (only leaf aspects can be concrete). An example of their work is the lacking option to override an aspect advice within this hierarchy, thus preventing its late binding at run-time. Similarly, overriding a pointcut designator might break inherited advice code. Briefly, the authors look at the options and restrictions posed by AspectJ with respect to object-oriented polymorphism of aspect classes.

Caesar [9] introduces the term "aspectual polymorphism" and Haupt et al. [10] show how Caesar and Steamloom aim at building "aspect-aware execution models". Given a hierarchy of a base and two derived aspect classes (B, A1, A2) and objects of both subclasses (a1, a2) it is possible to choose a specific instance of them by using the "deploy" keyword of Caesar. It allows to define an aspect variable being of type B and to assign a1 or a2 to it. Depending on the specific type of this variable instance, either advice code of A1 or of A2 will possibly run. Assuming the choice of A1, the advice code of A1 will run if some of the specified joint points (like method calls and executions) lie within the control flow of the deploy block. By the way, the aspect variable instance of the deploy statements might also be null, which means that no advice will be applied at all.

Aspectual polymorphism intends to enable/disable a selected aspect instance from a hierarchy of aspect classes and a set of their instances at run-time. Hence, this concept also focuses on aspect hierarchies and their instances. Enabling/disabling aspects aims at selecting the appropriate aspect instance and its advice in a polymorphic way (enabling/disabling an aspect instance at runtime is called dynamic aspect deployment).

In contrast, contextual polymorphism concentrates on the original classes and their methods. It considers the whole bunch of aspect advice potentially implying a behavioral change of an original method implementation. The actual selection of behavior is done at run-time due to added and enabled aspects and their advice. Whereas aspectual polymorphism deals with the appropriate dispatch of single aspect advice when enabling an aspect instance, contextual polymorphism covers all aspects and advice and the appropriate dispatch of a subset of them for each original method.

To summarize, aspects in the sense of AspectJ have only limited support of object-oriented polymorphism. Furthermore, the aspectual polymorphism approach of Caesar needs aspect deployment in order to make object-oriented polymorphism work for a hierarchy of aspect classes. On the other hand aspect deployment can be regarded as the basis of contextual polymorphism.

3.3 Implementation Approach of Contextual Polymorphism

In order to contemplate the relationship of contextual polymorphism and dynamic weaving we are going to describe the problems and a solution approach of contextual polymorphism.

3.3.1 Problem statement

First let us take a closer look at this new dimension of variation of a class method. For the sake of brevity we will concentrate on two aspects A and B (A precedes B), both having before, around and after advice for a specific class method m of class C. The dimension of variation can be seen as a linked list of method table blocks to be executed instead of the original code of C.m.

The next Figure should illustrate this more clearly. The block of A methods will start to be executed first calling *A.before* and *A.around*. The next block of methods (*B.before*, *B.around* and *B.after*) will run in case of *A.around* calling *proceed*. If *B.around* does not call proceed, C.m will not be executed at all. After *A.around* returns *A.after* will be called last.

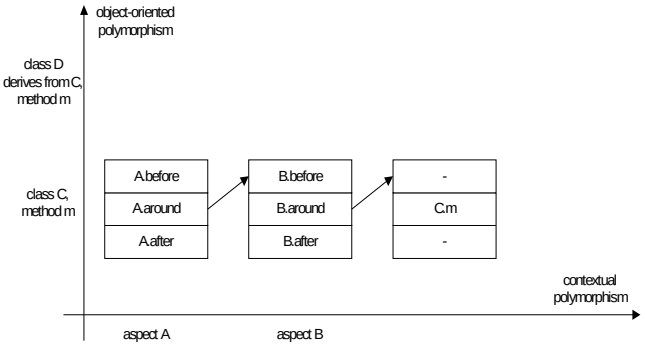

Figure 1. Contextual polymorphism with aspect precedence.

Looking at execution join-point as a start we have not found a way of inserting the complete sequence of advice calls into the original method body. For instance A.around requires B.before and B.after to surround its proceed statement in order to keep the advice flow correct. Therefore B.before and B.after have to be called in the execution context of A.around and defy their calling within the original method body. AspectJ solves this problem of handling around-advices by generating extra around methods specific to this join-point. This new around advice is responsible to call the next advice methods relevant to this join-point.

Thus, dynamic weaving alone does not make the generation of join-point specific advice methods obsolete. Taking dynamic enabling and disabling of aspects into account demands some more flexibility. Disabling aspect B makes it necessary to change the implementation of A.around. All invocations of advice code of aspect B has to be removed dynamically and replaced by a call to the original method code. Having a large sequence of aspects affecting C.m implies the extra effort of bookkeeping which join-point specific advice code has to call which other advice methods in order to locate and remove/add advice calls.

To sum up, using only dynamic weaving in case of hot deployment does not seem to make the generation of join-point specific advice methods redundant. On the contrary, these auxiliary advice methods need to be adapted dynamically to the changing aspect context of their join-point.

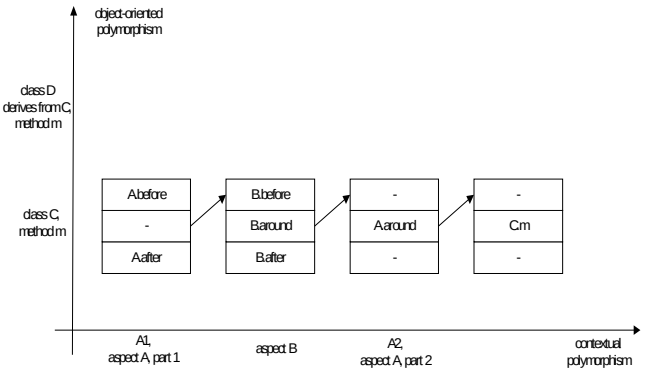

Figure 2. Contextual polymorphism without aspect precedence.

Last, let us investigate the case of no precedence specification between A and B. The corresponding variation can also be arranged linearly along the chain of method tables but it requires some adaptation in order to comply with the rules of AspectJ. All before methods of any aspect being relevant to C.m have to be executed before any around advice. This guarantees the execution of all before methods even in case of any around method omitting the call of proceed. This applies similarly to the after methods of all aspects relevant to C.m. Nevertheless, the same problems apply as in the previous situation covering a precedence declaration. Furthermore, a

solution has to cover the different control flow of advices concerning precedence and no precedence declarations.

3.3.2 Solution

Implementing this flow of aspect methods specific to C.m can be done by using delegation classes. For the purpose of comprehensibility we use A1_C_m, B_C_m, A2_C_m and C_m as different delegate classes. They have the common interface Delegate_C_m as shown in Figure 3.

```
interface Delegate_C_m
{ void proceed( Pointcut_Context_C_m ) };


class A1_C_m implements Delegate_C_m
{
 public A1_C_m( A a, Delegate_C_m next)
 { this.static_next = next; this.a = a;}

 public void proceed( Pointcut_Context_C_m c_m)
 {
   this.a.before( c_m.A_before );
   this.static_next.proceed( c_m );
   this.a.after( c_m.A_after );
 }

 private Delegate_C_m static_next = null;
 private A a = null;
}
```

Figure 3. Delegate class.

Classes B_C_m and A2_C_m have a corresponding structure and behavior. Thus C.m is associated to a chain of linked delegation classes. Assuming both aspects A and B being enabled at run-time the first delegate object is of class A1_C_m and has a relationship to an instance of B_C_m. In turn the delegate object of B_C_m is linked to an instance of A2_C_m. Additionally the Pointcut_Context_C_m class specific to C.m captures the whole pointcut context (e.g. arguments) to be made accessible to any aspect advice applied to C.m. This pointcut context is forwarded with any proceed call to the next delegate object to be able to retrieve the arguments of the aspect advice signature.

Thus we have described the maximum chain of delegation objects of C.m covering all relevant aspect advice to C.m. Although the order of all delegate classes can be resolved statically, we can not specify the type of the next delegation object of a delegate class like A1_C_m before run-time. This is due to the dynamic character of contextual polymorphism caused by enabling/disabling and adding/removing aspects.

Nevertheless, we need to keep track of the maximum chain of delegation objects at run-time. Disabling an aspect relevant to C.m could be done by removing the delegate object from the delegation chain. But subsequent enabling of this aspect has to reinsert the delegate object by keeping to the static order of aspect advice of the maximum chain. Therefore we have to keep both pieces of information within a delegate object: its static resolvable predecessor and successor of the maximum delegation chain and its current, dynamic predecessor and successor of enabled delegate objects of C.m as shown in Figure 4.

```
class A1_C_m implements Delegate_C_m
{
 public A1_C_m( A a,
                Delegate_C_m static_next,
                Delegate_C_m_static_previous ) …

 private Delegate_C_m static_next = null;
```

```
 private Delegate_C_m static_previous = null;

 // need to be thread-specific !
 private Delegate_C_m dynamic_next = null;
 private Delegate_C_m dynamic_previous = null;
}
```

Figure 4. Static and dynamic neighbours of the delegate class.

Consequently, there are two double-linked chains. The static chain of neighbors represents the maximum chain of delegation objects covering all aspects relevant to C.m, whereas the dynamic neighbors reflect the actual and dynamic chain of enabled aspects. Additionally the dynamic delegate members of a delegate class should be thread-specific since enabling/disabling aspects should be constrained to the current thread of execution only as described earlier.

Hence, it is possible to identify if a delegate object is active due to its dynamic successor (and predecessor) being null. Using both double-linked chains (the static and dynamic neighbors of delegate objects) also allows to search for the right position of the current chain of delegation a delegate has to be inserted into in case of being enabled.

The static neighbors have a further responsibility. Despite being of static nature they might change for the purpose of adding/removing aspects at run-time. This can be implemented by extending/reducing the maximum chain of delegate objects. The static neighbors of two delegate objects have to be adapted in case of adding a new aspect by inserting a new delegate object between these two statically linked delegate objects. The same holds in case of removing an aspect.

Consequently, delegation classes provide the necessary infrastructure in order to maintain the current and the maximum chain of aspect advice specific to C.m. Nevertheless, since delegates are passive entities we need an active execution environment (called *AOP Environment*) to trigger the necessary actions:

- instantiating delegate classes and creating the maximum chain of delegation of C.m,
- maintaining all delegate objects being associated to each aspect class/instance (since changing the aspectual context requires to retrieve all delegate objects to an aspect),
- controlling the chains of delegation objects,
- extending/reducing the dynamic chain of delegate objects in order to enable/disable aspects, offering an interface to enable/disable aspects,
- extending/reducing the maximum chain of delegate objects for the purpose of adding/removing new aspects, offering an interface to add new aspects and remove aspects,
- identifying all methods according to the pointcut declarations of new aspect advice and finding chains of delegation if already existent.

Having introduced the AOP Environment and its responsibilities we are able to address the issue which method code will replace the original method C.m. Basically it is a call into the AOP Environment to retrieve the current start of dynamically linked delegate objects of the respective chain of C.m. Besides it is necessary to create the Pointcut_Context_C_m object capturing the whole needed pointcut context of C.m.

Finally, we would like to mention that classes like A1_C_m, B_C_m, A2_C_m, C_m and Pointcut_ Context_C_m do not have to be generated for each method. There are some means to use generic classes instead of being dependent on classes and methods. Since these ideas are not necessary to convey the implementation approach of contextual polymorphism we do not further elaborate on them.

3.4 Contextual Polymorphism and Dynamic Weaving

Having glanced at some necessary parts of an implementation of contextual polymorphism we are going to examine its relationship to dynamic weaving.

In order to go on let us first clarify the term dynamic weaving. Basically, it means that operations like weaving and unweaving of aspect code into the base code it should be applied to are done at run-time. Nevertheless, by looking at the specifics of a weaving operation, it is useful to take a closer look at the earliest point of time of the running application this operation can be performed. We state that enabling/disabling aspects can be prepared to a very large extent at startup time, whereas adding/removing aspects requires weaving operations that can only be done while the application is running. In our opinion this is also connected to another facet of dynamic weaving: the identification of the execution points in the base code the aspect code should be applied to. If this identification can only be done at run-time after starting the application, some technical means of dynamic weaving at run-time are a necessity.

Steamloom in combination with Caesar's deploy keyword can be used to illustrate the enabling and disabling of aspects. Caesar uses the deploy keyword to enable an aspect. It also opens a code block within whose scope this aspect should be valid. This keyword can be implemented by using Steamloom's dynamic weaving capabilities.

Thus, enabling/disabling aspects essentially deals with a pre-known set of aspects. The base code and the aspects potentially affecting certain execution points in it (like methods) are well-known at startup time. Hence, it is possible to identify all aspect advice being relevant to each class method. Building up the infrastructure demands the instantiation of delegate classes and their linkage to a static chain of delegate objects. Since aspects will be enabled during the flow of the application, the dynamic chain of delegation of C.m only contains the delegate object of class C_m at startup. The dynamic neighbors of all other delegate objects of the chain are set to null. Furthermore, C.m has to be replaced by an equivalent method calling the AOP Environment in order to retrieve the first valid delegate object.

All these weaving operations can be done at startup. Operations like enabling/disabling aspects within code can be handled by using the AOP Environment which updates the relevant chains of delegation.

In comparison, adding new aspects with their pointcut specifications involves the execution of all the weaving operations described above at run-time since these potentially valid aspects have not been known at startup. These operations change the set of aspects and some dynamic delegate chains. Most probably completely new chains of delegation objects have to be created by the AOP Environment, too.

Finally, we would like to address the following argument against facilitating the enabling/disabling infrastructure of aspects at startup. Caesar blocks enabling aspects might not run during the execution flow of an application. Consequently, the argument goes that weaving should be delayed until actually entering the block of the "deploy" keyword at run-time.

First of all, weaving dynamically at run-time for a large extent of methods might imply some drawbacks with respect to performance and responsiveness of the system. Furthermore, regarding the analogy of contextual and object-oriented polymorphism let us investigate how object-oriented polymorphism handles this matter. Changing object references of a base class at run-time involves selecting the special virtual method implementation to be dispatched. But this is not done at run-time either. Rather virtual method tables facilitate this selection already at compile time. Thus it might be even more favorable to realize contextual polymorphism at compile time. At least this could work concerning the enabling/disabling of pre-known aspects.

But let us consider adding further aspects, too. New aspects eventually affect new methods which have not even been facilitated yet. It would be necessary to facilitate contextual polymorphism for all methods at compile time independent of its actual usage (see JBoss of

Section 2). This mechanism would compare to implicitly converting all non-virtual methods to virtual.

As a consequence realizing contextual polymorphism at startup and run-time leverages the affinity of these two times of execution and makes use of the same dynamic weaving techniques.

3.5 Contextual Polymorphism and Hot-Deployment

Now we are able to put all three terms “dynamic aspect weaving”, “contextual polymorphism” and “hot-deployment” into a sensible relation. Dynamic weaving represents a technique to implement contextual polymorphism. On the other hand contextual polymorphism is aligned along the notion of object-oriented polymorphism and comprises some theoretical thoughts with respect to a changing context of aspects at run-time. Hot-deployment takes both, dynamic weaving and the original code prepared for contextual polymorphism to add, remove, enable and disable aspects at run-time.

An analogy which does not comply exactly but isn’t far-fetched either expresses this relationship even more clearly. Dynamic weaving compares to an interpreter and execution environment transforming an aspect-oriented language that supports contextual polymorphism. Additionally the execution environment (e.g. the AOP Environment) allows to be accessed directly on code level in order to add/remove and enable/disable aspects. Hot-deployment harnesses the capabilities of such an interpreter/execution environment in order to add, remove and exchange specific cross-cutting concerns of an application.

4 What about .NET?

In the previous chapters we explained our meaning of hot-deployment of aspects and how our contextual polymorphism in combination with dynamic aspect weaving works. In this chapter we will explain the realization of our approach using the .NET environment.

Using the Microsoft .NET platform for implementing a dynamic aspect weaving tool has several restrictions, e.g. there is no possibility for exchanging the class loader. Since no open source implementation of the Common Language Runtime (CLR, the .NET counterpart to the Java Virtual Machine) was available when we started our work on .NET, we decided to use the Profiling and Metadata Unmanaged[1] API of Microsoft to realize our concepts. Additionally we use two libraries BARTO [11] and MEXlib [12]. The first one is a library to analyze, instrument, and/or construct IL code of a managed .NET method. The latter one is a library for the analysis, manipulation and extension of .NET metadata in unmanaged code. Using these libraries, it is possible to analyze existing types, create new types and change existing types at run-time. Compared to the *Reflection* and *Reflection.Emit* packages which are included in the .NET Base Class Library, MEXlib and BARTO work outside the CLR of .NET.

By the way, the libraries mentioned above (BARTO and MEXlib) and parts of the work to be described next can even be used with Rotor, an open source implementation of the CLR.

[1] Code that can be compiled to Microsoft’s intermediate language (IL) is called managed code in .NET because only .NET has complete control over such code. Memory allocation commonly used in C++ for example is a language feature that can not be mapped to IL, thus portions of code that contain such constructs are unmanaged code.

4.1 Components of Our Approach

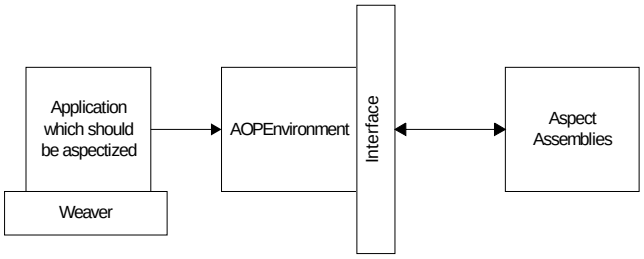

Figure 5. Components.

In our approach all components are managed by the CLR except the weaver (see Figure 5). In our realization a weaver is nothing else than a profiler and has to be registered to the application. Due to the uniformity of the words 'profiler' and 'weaver', it should also be mentioned that we will use them equally in the following. A profiler operates outside the CLR and is implemented as a COM component to be able to affect on the runtime. As the runtime environment is halted during the process of event notification no managed code can be executed in this state. Therefore both libraries mentioned above – MEXlib and BARTO – had been implemented using unmanaged C++. Basically, the profiling API was intended to be used for performance measurements of applications. It allows to register a component for events of the CLR such as assembly, module and type loading, JIT compilation of managed methods, garbage collection and throwing of exceptions. Currently, more than sixty different kinds of events of the CLR are available a profiler can be registered for and therefore it is the best intercepting mechanism to our opinion. When an application starts up, the profiler gets all events it is registered for. With the file that contains the weaving information, i.e. the information about which advice should be activated at which join point(s), the profiler knows how to instrument the code. For example if a specific type load starts, the profiler can introduce new methods, fields, etc. The application to be 'aspectized' has to reference the AOP Environment (a C# class library) so that the profiler is able to weave in a call to the AOP Environment. As already mentioned in chapter three, our AOP Environment offers two interfaces in order to add/remove and enable/disable aspects from outside. Besides, the AOP Environment has to offer some triggering mechanism to signal that aspects have to be added/removed independent of the running application. This can be for example a communication channel listening to commandos of a graphical user interface (GUI). This GUI identifies the location of new aspect assemblies, the corresponding weaving information and an enumeration of aspects to be removed. As a result of such a triggering mechanism, the application can be implemented without knowing anything about the AOP Environment.

4.2 Weaving

Our approach differs from others in the way that it does not weave aspect code into business code directly. The only thing to be woven into the business code is the call to the AOP Environment. As already mentioned in chapter three, the AOP Environment is responsible for executing all advice code of the currently active aspects. The weaver has to decide how and where to weave and unweave a call to the AOP Environment into a method. For the ease of understanding we will explain our approach using an example. Imagine a class C containing one method m and an aspect A containing an advice which is executed before every execution of method m. It should be mentioned that in our approach aspects are treated as ordinary classes and pieces of advice as methods. Every time method m is executed and aspect A is active, the advice code of aspect A is executed before the original code of m. Our weaver copies the method body of C.m to a temporary generated method e.g. m_aspectized, and weaves in the call to the AOP Environment

into the original method C.m (see Figure 6). When the methods are executed the first time, they will be JIT compiled automatically.

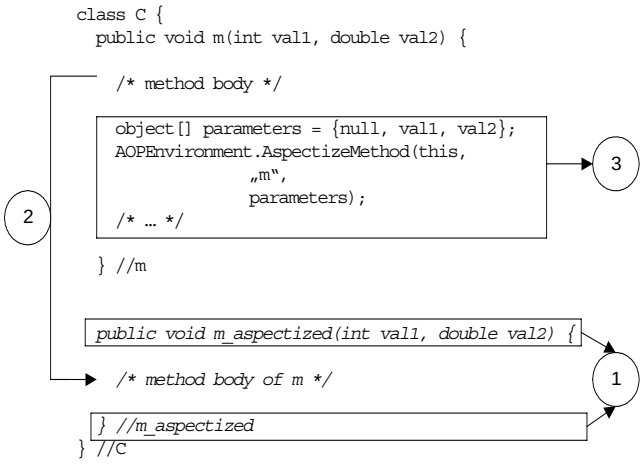

Figure 6. Aspectizing the execution of a method.

The call to the AOP Environment *AOPEnvironment.AspectizeMethod* has three parameters. The first one is the current object – to be able to call the original method at the end of the chain of delegation. The next one is the name of the method that should be executed after all aspects have been applied to the method which is 'aspectized', and the last one is an array which contains the return value and all parameters of the current method.

4.3 Enabling and Disabling Aspects

Enabling (disabling) of aspects can be realized by adding the delegation objects into the respective dynamic chains of delegation if existent. If there is no delegation chain for a specified method yet, a new one is created and the delegation object is added to it. This functionality is provided by an interface of the AOP Environment. The interface contains two methods, one for enabling an aspect and one for disabling an aspect. Both methods take the aspect name as argument.

4.4 Adding and Removing Aspects

To complete the hot-deployment of aspects, the AOP Environment offers another interface. It contains a method for adding new aspects (*AddAspect*) and one for removing aspects (*RemoveAspect*). The first one takes two parameters, the path to the aspect assembly and the path to the weaving information. The latter one expects as argument only the name of the aspect to be removed.

When *AddAspect* is called, the assembly located at the given path is loaded using the .NET reflection method *Assembly.LoadFrom(string assemblyFile)*. The aspects and delegate objects are instantiated and put into the respective delegation chains as mentioned above. When calling the *Assembly.LoadFrom* method, the profiler gets – some settings presumed – several events from the CLR like *AssemblyLoadStarted*, *ModuleAttachedToAssembly*. If parts of the loaded assembly's name match a pre-defined prefix like "AspectAssembly_", the profiler knows that one or more aspects should be added. Now the weaving information is processed by parsing it. Every time a pointcut occurs in the weaving information file, the profiler looks up an internal data structure, to see whether the necessary calls to the AOP Environment already exist. If they do already exist, nothing has to be done. Otherwise, a process is started to weave in the calls to the AOP

Environment. This happens by marking the necessary methods to be rejitted. Before a method is JIT compiled again it is instrumented, so that the AOP Environment is called the next time the method is called.

Calling *RemoveAspect*, the AOP Environment has to weave out an aspect. This process works similar to adding aspects. The weaver looks up its internal data structure to find all join points affected by the aspect. If the aspect is the only one registered for a specified join point, the call to the AOP Environment will be woven out by marking the necessary method for being rejitted. The next time the method is JIT compiled, the instrumented method body is replaced by the original one. If more than one aspect is registered for a specific join point, nothing will happen as the call still has to persist for the other registered aspects.

Based on an AOP Environment that has been a result of a diploma thesis [13] at Siemens, we are in progress of developing our prototype. The next steps are the design and implementation of a join point model and the reengineering of the existing AOP Environment. Both are topics for further diploma theses.

5 Summary

We have started our investigation of dynamic aspect weaving with two examples from the domain of automation and telecommunication. These examples motivate the need of hot-deployment of aspects, especially the exchange or the customization of cross-cutting concerns at run-time. In our opinion hot-deployment comprises the option to enable and disable aspects of pre-known aspects a run-time. In addition, it includes some means to add totally new aspects and their pointcut specification during a running application (and to remove them).

As a consequence we have derived a first notion of contextual polymorphism underlying the specifics of hot-deployment. Contextual polymorphism relates to object-oriented polymorphism in the way that it overrides class behavior. Since the contextual set of aspects changes by enabling/disabling and adding/removing aspects at run-time, each method implementation of a class might be overridden by different aspect advice at separate points of execution. Therefore, we have based the term contextual polymorphism on the similarities of contextual to object-oriented polymorphism. Nevertheless, we have addressed some differences to object-oriented and aspect(ual) polymorphism.

Putting contextual polymorphism into relation to dynamic weaving we have stated that contextual polymorphism necessitates dynamic weaving due to adding and removing aspects at run-time. Enabling and disabling aspects can be largely achieved by facilitating their usage at startup-time. Finally we have mentioned the specifics of contextual polymorphism and dynamic weaving on the .NET platform which historically was the starting point of our work on aspect-oriented programming.

References

[1] AspectJ Home Page. http://www.eclipse.org/aspectj.

[2] JMangler Home Page. http://javalab.iai.unibonn.de/research/jmangler/

[3] JMunger Home Page. http://wwwhome.cs.utwente.nl/~bakkerj/jmunger/

[4] W. Schult and A. Polze. Dynamic Aspect Weaving with .NET. www.dcl.hpi.unipotsdam.de/cms/papers/papers/GI2002.ps

[5] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann. *Virtual Machine Support for Dynamic Aspects.* To appear in Proceedings of 3rd International Conference on Aspect-Oriented Software Development, 2004.

[6] JBoss AOP Home Page. http://www.jboss.org/developers/projects/jboss/aop.jsp.

[7] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. *JAC: A flexible solution for aspect-oriented programming in Java*. In Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns pages 1-24, 2001.

[8] E. Ernst, D. H. Lorenz. *Aspects and Polymorphism in AspectJ*. In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pages 150-158, 2003.

[9] M. Mezini, K. Ostermann. *Conquering Aspects with Caesar*. In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pages 90-100, 2003.

[10] M. Haupt, C. Bockisch, M. Mezini, K. Ostermann. *Towards Aspect-Aware Execution Models.* Technical Report, 2003.

[11] M. Umgeher. *BARTO - Bytecode Analysis, Representation, and Transformation Objects*. Diploma Thesis in Software Engineering at the Upper Austrian University of Applied Sciences, September 2003

[12] F. Schmied. *Design and Implementation of MEXlib*. Diploma Thesis in Software Engineering at the Upper Austrian University of Applied Sciences, July 2003

[13] M. Schüpany. *Entwurf und Implementierung einer Entwicklungsumgebung zur aspektorientierten Programmierung für die .NET Plattform*. Diploma Thesis in Software Engineering at the Upper Austrian University of Applied Sciences, June 2002

Comparing Dynamic AO Systems

Ruzanna Chitchyan
Ian Sommerville
Computing Department, Lancaster University, Lancaster, UK
{rouza,is}@comp.lancs.ac.uk

Abstract

In this paper we present a comparative analysis of several currently available Java based dynamic AO systems. The comparison is built on how these systems deal with general dynamic reconfiguration problems. Through this exercise we hope to better understand weather and how do the specific AO change support mechanisms (e.g. total, actual, or collected weaving) affect dynamic system reconfigurability.

1 Introduction

Many software systems remain in operation for several decades. Among them there is a class of systems which is required to be continuously operational, as their interruption will result in economic loss (e.g. telecommunications systems), environmental damage (e.g. nuclear plants) or even loss of human life (e.g. life support systems). The maintenance and upgrading of such systems have to be carried out without shutting the systems down, i.e. dynamically.

We call a system *dynamic* if it provides support for changing its organisation as a concurrent activity to the application providing services. We also say that a system *supports dynamic applications* if the organisation or functionality of applications based on the system can be changed without the application being interrupted. While both of these properties could combine, providing a *dynamic system supporting dynamic applications*, each of them could also exist independently. Furthermore, a system is a *dynamic aspect-oriented system* if it additionally accommodates dynamic change with *crosscutting* concerns (i.e. concerns simultaneously affecting multiple system or/and application units).

We maintain that the issue addressed by all dynamic systems is that of dynamic reconfiguration, however in the context of aspect-oriented software development the traditional solutions of configuration paradigm based on components, ports and bindings are unsuitable. Traditionally, components explicitly provide ports for binding, i.e. are pre-designed for a specific composition, while aspects cannot always expect pre-planned design support. Consequently, research in dynamic AO has resorted to alternative mechanisms.

In the present paper we look at a Java-based subset of dynamic AO systems, Java being currently the most widely used language in the AO community. Generalising from this set of systems, we identify some alternatives for dynamic change support mechanisms, depending on (a) when the change is incorporated (or *woven*) and (b) how the weaving is accommodated.

We also present a set of generic criteria for dynamic reconfiguration and evaluate the selected systems with respect to it, highlighting how the systems differ, depending on their change support mechanisms. Through this exercise we hope to better understand weather and how do the specific AO change support mechanisms affect dynamic system reconfigurability.

The rest of this paper is structured as follows: section two discusses the above mentioned change support mechanisms in more detail, section three presents the generic reconfiguration criteria, section four discusses how our selected systems compare with respect to the introduced criteria, finally section five provides summaries and conclusions.

2 Change support criteria[1]

Having found traditional reconfiguration solutions unsatisfactory, dynamic AO research with Java has resorted to byte-code modification (e.g. [1]) and reflection based (e.g. [2]) solutions. Depending on what stage change is woven into base programme, these solutions can be classified as using:

- Load-time weaving;
- Just in Time (JIT) compiler weaving;
- Dynamic proxies.

Load-time weaving based systems perform byte code transformation at *class loader* level: the code is transformed as it is loaded into the VM. The class loader based approaches either subclass from the Java abstract ClassLoader class – this is a standard Java mechanism – or completely replace the root class loader (JMangler [3], AspectWerkz [1] approach) the later however breaches the Java security mechanism. When sub-classing is used, the *problem of advising system classes* arises. This is caused due to Java security mechanism, where applications loaded by user-defined class loaders are prohibited access to Java system classes loaded by the system class loaders. Moreover, the namespace visibility constraints enforced by class loader hierarchies present additional problem when a crosscutting concern needs to be able to access independently loaded modules. Due to all these tradeoffs often custom class loaders, deviating from standard security mechanisms, are preferred (e.g. JBoss [4]).

With JIT compiler level modification the bytecode is loaded into the VM without transformation. The alteration takes place when the JIT compiler compiles the bytecode into a native code. Consequently, for this approach, the JIT compiler needs to be augmented with additional functionality.

The Dynamic Proxy[2] approach is solely a Java Reflection based solution: no code transformation takes place, only standard Java language mechanisms are used.

Both class loader and JIT based weavings require modification of bytecode. Depending on *how* the code is transformed to accommodate change, the systems can be further classified[3] as those using:

- *Total* hook weaving: augment the entire code at each possible join point with a hook to which additional behaviour could reference;
- *Actual* hook weaving: weave hooks only to a set of points of actual interest, not to every possible point of potential interest;
- *Collected* weaving: weave in the code (rather then hooks) for additional behaviour at the points of interest, with the resulting code collecting the aspects and base in one unit.

Each of the above alternatives has its strengths and weaknesses. JIT, for instance, could be used for both *collected* and *hook weaving*. While the *collected weaving* will improve the performance[4] of the AO system (due to reduction of number of indirect references), it will also tightly bind the aspect and base code, making *unweaving* for advice removal more complicated.

---

[1] Much of the discussion in this section is synthesized from documentation and publications for the approaches discussed in section 4 of this paper.

[2] A dynamic proxy is a class generated at runtime that implements one or more separate interfaces. A call to the methods of an instance of the proxy will be re-directed to an object implementing InvocationHandler.

[3] Adopted from [13-14].

[4] Assuming that inlined code is executed more than ones, or the inlining takes lesser time than indirect referencing.

The *hook weaving* options, on the other hand, are expensive from the performance point of view, but preserve separation of aspect and base code and simplify future attachment/detachment of aspects to any potential point of interest without need for repeated base code transformation.

The summary table below grades the various weaving options, with A given for best and C for worst results:

Table1. Summary of tradeoffs for alternative weaving approaches

criteria / Weaving approach	Evolvability	Excessive Code	Performance
Total hook weaving	A	C	C
Actual hook weaving	B	B	B
Collected weaving	C	A	A

It is worth noting that in practice a combination of the discussed mechanisms could be used in a single system.

In this section we have discussed several alternative implementation mechanisms, but how do these mechanisms help to address the dynamic reconfiguration problems? To address this question, we first define a set of generic reconfiguration criteria in the following section, then evaluate systems with the above implementations against these criteria.

3 Criteria for comparison

The criteria presented below are gathered from a body of work on dynamic reconfiguration [5-9], which, as discussed earlier, is directly related to the problems being addressed by dynamic AO systems:

1. *Separation of application functionality from structure* (or *level of coupling*) determines whether the software system can be defined in terms of loosely coupled units. If yes, then the developers can have 2 different – functional and structural – views on it. The functional view pertains to the state and behaviour of a unit, while the structural view presents the system as a graph of interconnected components (with no care about their functionality).
2. *Preservation of application integrity* is self-evidently concerned with integrity preservation. This criterion can be further unravelled into *change action synchronisation* and *persistent state preservation*. The action synchronisation should ensure that the components involved into the reconfiguration process are made mutually consistent, and persistent state preservation is to ensure that any unprocessed messages of the changing component, and critical application data for the application survives the reconfiguration.
3. *Application contribution to the reconfiguration process* defines how much does the application need to contribute towards the system reconfiguration process. While ideally the application will be completely unaware about reconfiguration, in reality it could need to perform some activities to ensure that its correctness is preserved.
4. *Reconfiguration specification* considers how are the required changes presented. Ideally this will be a declarative specification, telling what needs to be changed, rather than how to do the changes. This criterion contributes towards understandability of the changes and their analysability.

5. *Efficiency*: this can be broken down to *application disturbance* and *time delay*. The application disturbance can be measured in terms of number of components affected by the reconfiguration.
6. *Programmed change robust to evolution*: where both programmed and evolutionary changes coexist, the programmed change must minimise its assumptions about the existing configuration, as the configuration could be unpredictably changed due to evolutionary requirements.

Although it might seem that AO systems should have additional reconfiguration criteria to address due to the wider impact and unanticipated nature of aspects, we are of the view that the above criteria are sufficiently broad to accommodate aspect-specific issues. This is because other reconfiguration actions could potentially have wider impact on system modules (e.g. structural change) or cause unexpected interactions, as aspects could.

4 Systems for comparison

The systems selected for evaluation in this paper are AspectWerkz, JBoss, PROSE, and Nanning. While all these systems are considered to be dynamic, they provide varying level of dynamic change support. A brief introduction to each of these systems is provided (section 4.1) followed by a discussion on how they perform against the dynamic reconfigurability criteria (section 4.2).

4.1 Outline of the systems

AspectWerkz [1, 10, 11] is a dynamic AOP framework for Java that uses *load time actual hook weaving* with custom-enhanced core java class loading architecture hooked in directly after the bootstrap class loader. It can perform bytecode transformations on classes loaded by all but the bootstrap loaders[1]. The framework has several architectures to support different versions of Java, however the recommended version (that for Java 1.4 release) is presented in the Figure 1[2].

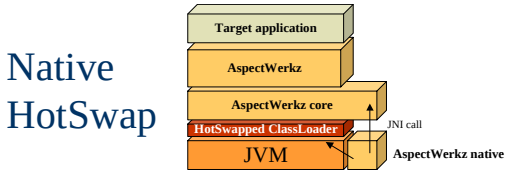

Figure 1. AspectWerkz Native HotSwap for Java 1.4 architecture. The JVM is launched with a native JVMPI extension. This extension, when loaded, hotswaps the java.lang.ClassLoader to AspectWerkz core and native JVMDI API. Thus, the system uses a single JVM, and a non-intrusive way of plugging in AspectWerkz in Java 1.4.

JBoss [4, 12] is a reflective and reconfigurable Java application server [4] which uses the JBoss AOP framework – where aspects are implemented as interceptors – to provide a set of crosscutting services (e.g. persistence, remote access etc.). JBoss uses *load time weaving* with a custom class loader (called *unified class loader*) that:

- loads all classes into a flat namespace, for the components to be able to share non-system classes, and

[1] The core *theoretical concept* of hooking in the class loader hierarchy is inspired by the JMangler project, but more recent versions of AspectWerkz (after 0.8) do not use JMangler at all.

[2] This figure is used from [11].

- inserts hooks into the bytecode to allow interceptor attachment for interception for field, constructor, and method manipulation.

By default JBoss uses *actual* hook weaving, in which case it is not possible to hot-deploy any AOP constructs to the units with non-augmented joinpoints. However, the system also provides an option for *total* hook weaving for any unit marked as `advisable`.

The general architecture of JBoss is presented on Figure 2.

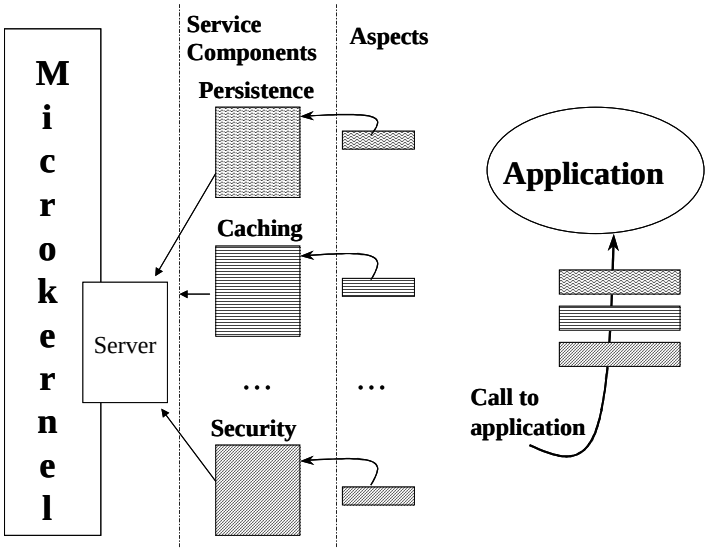

Figure 2. The JBoss architecture. The system consists of a *microkernel layer* which provides most of the "application server" functionality; a set of *deployable services* that can be dynamically added or removed, as required; an aspect layer where aspects are implemented as interceptors; and an application layer which contains the user applications written in plain Java [12]. The aspect layer is used to link the application layer to the JBoss services, allowing container type functionality to be added to plain Java objects. Both *dynamic proxies* and *interceptors* are used in this process.

The **Prose** [13-17] framework uses *JIT compiler weaving*, and allows both *total* and *actual* hook weaving. The weaving scope is specified through a (pattern of) class names and a specific Boolean variable, if the variable is set to false only the classes matching the given name (pattern) are augmented, otherwise these matching classes are left unchanged, and the rest of the code is augmented.

The system consists of two main layers – the Execution Monitor and the AOP Engine – demonstrated on Figure 3.

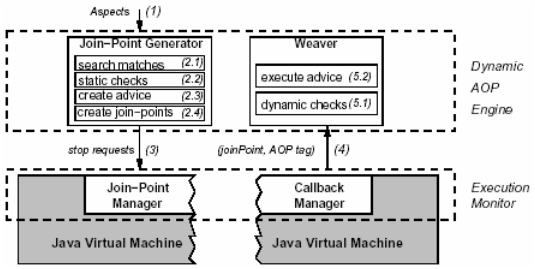

Figure 3[1]: Prose architecture. In the upper layer the AOP Engine accepts aspects (1) and transforms them into basic entities like joinpoint requests (2.1-2.4)[2], then activates these joinpoint requests by invoking methods of the Execution Monitor (3). The Execution Monitor is integrated with the JVM and serves to activate joinpoints at JIT time and to notify the AOP Engine when a joinpoint has been reached (at run time (4)). When notified, the Engine then executes the advise (5) for that joinpoint.

Although Prose can be extended by adding new *types* of joinpoints, it can only be done statically and requires quite a number of changes all across the system.

The **Nanning Aspects** [2] framework uses the *Proxy* facilities of Java Reflection API (Figure 4) and interceptors to provide aspect-oriented-type functionality at run time. Aspectised objects in Nanning consist of sets of interfaces, target objects, and interceptors. The interceptors are used for *around-advise-like-function*, i.e. an interceptor is called when the method is called and is responsible for further propagation of the call either to other interceptors, or to the initially called method. Currently only method-call pointcuts can be advised.

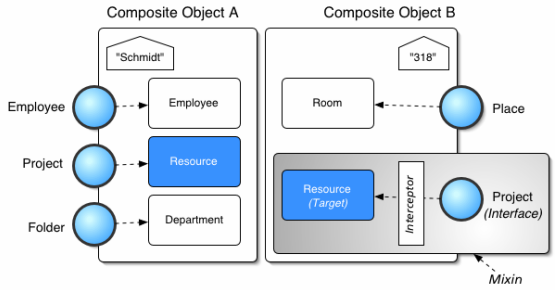

Figure 4[3]: Objects in Nanning.

No language extensions are used for any of the discussed systems.

4.2 Comparative Evaluation per Criteria

In the following discussion the systems themselves, as well as applications build with these systems are discussed.

[1] Figure reused from [13].

[2] 2.1 – inspect all the currently loaded classes and gather all matching requests (e.g. methods with given signature); 2.2 – perform static check (e.g. given method exists etc.); 2.3 – define the client data to be passed back by the weaver then the joinpoint is actually reached (e.g. the advice); 2.4 – generate joinpoint requests.

[3] Figure reused from [2].

4.2.1 Separation of application functionality from structure

The **AspectWerkz** framework itself is dynamic only in so much as the java.lang.ClassLoader is hotswaped to AspectWerkz core at framework load time. After this change the framework remains static; so the *framework itself* is not dynamically reconfigurable. It does however provide a certain set of *dynamic application support* features: advices can be added, removed and re-structured and the implementation of introductions can be swapped at runtime.

In AspectWerkz applications the advice and introductions are written in plain Java and their target classes can be regular plain Java objects . Aspects can be defined using either an XML definition file or Runtime Attributes. In the recent version of the framework (AW 0.9 release candidate) an aspect with its related pointcuts, introductions and advice can also be defined within a single aspect class. Whichever way aspects are defined, they need to be referenced to from within an XML[1] definition file that states which aspects apply to which classes. It should be noted that until recently no new aspects or pointcuts could have been introduced into a running application, neither the existing ones could have been removed, but the advice and introduction defined on existing pointcuts could have been taken out, replaced or updated. However, the very recent work on this system [18] is building towards allowing dynamic aspect deployment, though the issues of un-deployment have not yet been addressed.

Thus, at runtime, aspects are tightly coupled to the application and pointcuts are tightly coupled to the aspects in which they are designed. On the other side, the coupling of advice and introductions to the aspects is relatively loose.

JBoss middleware components are based on *Java Management Extension (JMX)* standard [19], which defines an architecture for dynamic management of recourses. As a result, each service, managed through JMX, can be added/removed from the running server without affecting the rest of the system. Thus, unlike AspectWerkz, *JBoss itself* is a *dynamic system* with well-defined structure and encapsulated functional modules where AOP framework is also a deployable module.

Applications developed with JBoss AOP are also dynamic, in that at runtime they can obtain, discard, and re-structure services they use through the JBoss AOP framework. In the AOP framework the advice are implemented as interceptors, introductions provide new interface types which can be implemented either from within an interceptor, or through a mixin class, with pointcuts defined in XML configuration file and no distinct entity for aspect concept to be mapped to. Due to this structure, changes in application structure - caused by attaching or detaching interceptors - are separate from the application code and reflected only in the corresponding XML configuration file.

As presented in Figure 3 for **Prose**, the Execution Monitor is embedded in the JVM while the AOP Engine is application specific. The Execution Monitor provides a Model for its JoinPoint API to the AOP Engine, so different engines could be used with the same monitor. Nevertheless, to the best of our knowledge, dynamic replacement of the Engine is not supported, and neither is the run-time change of the selected weaving mode. Thus, the system itself is not dynamic, but does provide *dynamic application support*: aspects can be woven and unwoven to/from the running system.

Aspects and Crosscuts – constructs containing advice and pointcut type information – are Java classes extending provided framework classes. These constructs are loosely coupled to the base application as their joinpoints are extracted and (un-) registered for interception at load time with no explicit binding requirements.

The runtime mechanism in core **Nanning** is entirely based on dynamic proxies. The *AspectInstance* class implements this runtime support for AO by containing aspects which *introduce* and *advice* the *AspectInstance*, as well as handling all calls to aspectised objects

[1] Depending on which aspect definition style is used, this file could contain more or less additional information.

(through its implemented *InvocationHandler* interface). Here *introductions* are implemented as mixins, *advice* as interceptors, and proxies are used to insert interceptors at run time between the call and the actual execution of a method. Pointcuts are instances of *Pointcut* class, defined and stored in aspects. This approach decouples the object implementations from their provided interfaces, since all references to an instance are directed to a proxy.

In short, summary of discussion for these criteria is presented below:

Table 2. Summary for criterion 1: Separation of application functionality from structure.

AspectWerkz	**JBoss**	**Prose**	**Nanning**
Framework itself is not intended for dynamic reconfiguration. Aspects are coupled to the application and pointcuts to the aspect but advice and introductions are loosely coupled to aspects.	JBoss itself is a dynamically reconfigurable system with loosely coupled functional modules. Interceptors, Introductions and mixins are loosely coupled to the applications, but closely dependant on pointcuts and so XML definition files.	Prose framework itself is not dynamically reconfigurable, though its main two components (AOP Engine and Execution monitor) are relatively loosely coupled. Prose aspects are loosely coupled to the base application, but Crosscuts are tightly coupled to aspects.	Proxies in Nanning help to decouple the object implementations from their provided interfaces. Pointcuts are coupled to the aspects, which in turn are coupled to AspectInstance and AspectSystem classes.

4.2.2 Preservation of application integrity

Since for AspectWerkz the class loader hotswap is the only dynamic step in the framework configuration, and that step is carefully designed and synchronised, the integrity of the framework itself cannot be jeopardised.

For the applications based on the framework the following features are helpful in preserving their integrity:

- both advice and introductions are synchronised (except when an introduction provides a marker interface with no implementation);
- both advice and introduction updates are generally thread save;
- default implementation of *JoinPointController* provides clear instructions for *consecutive* execution of each advice/introduction for each join point;
- default implementations for advice and introduction persistence are provided.

The first three points above support *change actions synchronisation*, and the last point is helpful for *persistent state* preservation.

On the other hand, there are no in-built measures for synchronisation of the application state *as a whole* or its persistent state when replacing or updating advice and introduction implementations[1], except the requirement that the replacing introduction implementation has to implement the same interface as the replaced one.

The AOP services in **JBoss** currently provide a set of pre-packaged crosscutting functionality (e.g. transactions, security, etc.) which are additive in nature and so do not interrupt the existing functionality of the applications. All the updates are addressed "in one action",

[1] Suppose an advice has a state variable defined, when it is replaced with a new advice, the variable in the new advice will not be re-initialised to the value of the one being updated.

triggered by the update of the changed configuration specification file. Furthermore, since objects are not aware of interceptors, they do not need to be adjusted for their change. However, guards for preserving semantic integrity of applications need to be incorporated in interceptors (e.g. persistence should not be introduced half-way through a transaction, etc.). Although in the provided set of JBoss AOP services these guards (e.g. threading and synchronisation) could be taken care of, a wider use of AO for non-standard processing could become conflict-prone in this respect.

One of the requirements for the **Prose** system during its development was "secure and atomic weaving". The atomicity is supported through blocking the advice execution in the hook methods till the weaving operation for the whole system is completed.

Atomicity of weaving ensures simultaneous update of affected units. An option for transactional weaving for several (un-) weave operations is also provided, allowing weaving operations to be prepared, but not committed until a separate commit instruction. This option could be used, for instance, to synchronise weaving on distributed systems, or to indirectly assist in persistent state preservation. While persistent state preservation is not supported explicitly, a developer aware of potential persistent state violation, can provide specific checks before committing (several) changes. For instance, suppose "withdraw from account A and credit to B" is being processed; after the withdrawal has been executed a transaction support aspect is dynamically woven in. Due to atomic weaving all affected parts of the application will acquire transaction support from the point of the weaving onwards, but for the completeness and correctness of the first semantic transaction the withdrawal needs to be accounted for as well. Via the "commit" operation, the developer will be able to instruct the committing of transactional aspect, for instance, after completion of the started semantic transaction.

On the other hand, since the dynamic support for Prose allows only for weaving and unweaving of aspects in transactional manner, the issue of integrity of aspects themselves does not raise.

Since in **Nanning Aspects** all references to an object instance are directed to a proxy, an object implementation can be changed during runtime with all its references still remaining intact. Thus change of advice or introduction implementations will not affect the applications as long as the interfaces remain in use. Besides, the Proxy is a Serializable class which is helpful for change action synchronisation.

The summary of discussion for this criterion is presented in Table 3.

Table 3. Summary for criterion 2: preservation of application integrity.

AspectWerkz	JBoss	Prose	Nanning
The integrity of framework itself is well preserved. There is also some support for change synchronisation and persistence in applications, but a holistic support mechanism is missing.	Pre-packaged aspects can cope with change synchronisation and persistent state preservation, however the wider use of AO could be conflict-prone if integrity guards have to be cared for in interceptors.	Since insertion and withdrawal are performed through registering and un-registering points of interest to events, addition or removal of aspects does not significantly disrupt the application, but there is no explicit support for persistent state preservation.	Provides some support for synchronisation of change and persistent state preservation.

4.2.3 Application contribution to the reconfiguration process

As already mentioned, in the **AspectWerkz** framework the application has a significant role to play in preserving its integrity. In particular since there are no semantic integrity preservation constraints at the framework level, these must be provided at the application level. To achieve

this the application might need to provide custom implementations for a number of default features, such as:

- *JoinPointController* concerning business logic for handling e.g. advice redundancy, dependency, or compatibility, or special exception handling.
- pluggable container, e.g. for an application-specific persistence policy;
- in cases when a thread is being hotswapped into an advice, a special method for thread resumption needs to be called manually;
- when replacing an introduction, the new one is required to implement the same interface as the one being updated.

Since the intention of the **JBoss** AOP is to augment the application with additional functionality without its knowledge, the application should not be required to provide any contribution at all. However, in reality the application sometimes needs to provide certain support for dynamic aspect reconfiguration. For instance in order to be able to make an object remotely accessible through JBoss Remoting aspect, the object to be aspectised must have a default constructor in its class definition and also the parameters and return values for the remotely invoked method must be Serializable, etc. These and similar issues must be explicitly dealt with by the applications.

In **Prose** the weaving and unweaving are transactions. However, as already discussed in the example for *preservation of application integrity* subsection, the system does not explicitly address the issues of persistent state preservation and the application programmer should provide appropriate checks at the application level.

Although **Nanning** uses some mechanisms for change synchronisation, applications could be required to contribute to the reconfiguration process if, for instance, some private non-persistent data needs to be processed before change.

The summary of discussion for this criterion is presented in Table 4.

Table 4. Summary for criterion 3: application contribution to the reconfiguration process.

AspectWerkz	JBoss	Prose	Nanning
The application has a significant role to play in the reconfiguration process, triggered by need to preserve its integrity.	The applications need to provide certain support for reconfigurability with aspects.	The applications need to provide certain support for reconfigurability with aspects.	The applications might be required to provide certain support for reconfigurability with aspects.

4.2.4 Reconfiguration specification

In **AspectWerkz** the change specification could be considered imperative, as the steps to be taken need to be provided in a prescriptive pieces of Java programme. And although all used methods are provided by the system, part of the specification requires some additional coding (e.g. how to re-order the advice applied to a pointcut.).

In **JBoss** change specification is provided through XML files, thus it is declarative. XML is easily analysable as well as understandable to the readers. Not only change specification is provided via XML, but also the pointcuts and class metadata. While it is helpful to have all this data available in one place, the file could become very large and so less readable.

The types of change supported by **Prose** are aspect weaving and unweaving. Both of these changes can be specified declaratively either through a graphical user interface of the Prose

WbProse tool, or through command line tool for both local and remote weaving/unweaving operations.

Configuration of **Nanning's** aspects is provided either externally in a declarative XML file or in Java code and via run-time attributes. While XML configuration allows to locate everything in a single file, it also could become very large with large portion of the system's behaviour being defined in it. Nanning documentation [2] cautions against this "XML hell", suggesting partial use of imperative in-code configuration.

The summary of discussion for this criterion is presented in Table 5.

Table 5. Summary for criterion 4: reconfiguration specification.

AspectWerkz	JBoss	Prose	Nanning
Could be considered imperative as some "how to change it" code could be required, though some declarative XML is also used.	Declarative, via XML.	Declarative, via set of simple commands (e.g. insert) or GUI-based tool.	Could be a mix of imperative and declarative styles.

4.2.5 Efficiency

In **AspectWerkz** the disturbance due to dynamic change is limited to actually updated advice or introductions, with no effect on already loaded code.

The overhead of bytecode modification is rather light when no class has bound aspects, or binding occurs only once per class in each class loader hierarchy. The documentation [1] suggests that an advice or introduction adds an overhead of only ~ 0.00025 ms per call[1].

In **JBoss** the additional bytecode added by the instrumentation for interceptor attachment have some time delay hit. This could be minimised by fine-tuning the AspectManagerService mbean (which is pre-defined to instrument all possible points for pointcuts) to disable instrumentation of some unused pointcuts (e.g. all Filed access points). However, this will disallow use of interceptors for non-augmented points, and if their use is required later on, the base code will have to be re-augmented. When no re-transformation is required, disturbance due to interceptor attachment/detachment is rather limited.

In **Prose** the disturbance measure is high for 2 reasons:

- due to transactional nature of weave/unweave operations, all affected units in the system as well as units that will attempt to use them, will be prevented from progression until the completion of the change operation.;
- since there are no other update operations than weave/unweave for the whole aspect, every change to the woven aspects will require unweaving and reweaving, thus affecting all units advised by the changing aspect.

On the positive side, Prose provides both run-time and insertion-time filtering. The insertion time filtering is used to prevent joinpoint activation during aspect insertion. This is a more efficient way of filtering, as no run-time overhead will be introduced due to non-required joinpoint activation and evaluation at run time. The *response time* of dynamic weaving is also negligible [13], thought PROSE is relatively slow, compared to other approaches.

In **Nanning** disturbance will be limited to the currently updated aspectised object. And although here, like in AspectWerkz, *actual hook weaving* is used the response time will be significantly higher due to extensive use of reflection and presence of the proxy layer.

The summary of discussion for this criterion is presented Table 6.

[1] Measured on Pentium 4 2.56 Mhz, 215 RAM.

Table 6. Summary for criterion 5: Efficiency.

AspectWerkz	JBoss	Prose	Nanning
Limited disturbance and good response time.	Disturbance is limited, but there is a trade off between disturbance and response time.	Rather high disturbance, but negligible response time for dynamic weaving.	Limited disturbance but high response time.

4.2.6 Programmed change robust to evolution:

Ideally, an aspect should be able to introduce a change into the base without any undesirable side effects. However, by now it is well known that some issues of the AO technology make it difficult to guarantee side-effect free evolutionary aspectisation. Such issues arise, for instance, due to use of generic patterns (e.g. with wildcards) in pointcut specifications[1] or unintended interaction between aspects applied to the same joinpoint, etc. These generic pitfalls are present in all of the considered systems, as all of them use, for instance, patterns for pointcut specification.

Applications developed with the **AspectWerkz** framework support change in terms of addition, removal and updating of advice and introductions to already defined pointcuts, and more recently addition of new aspects with their pointcuts. **Prose** supports change through registration or change of pointcuts and dynamic weave/unweave of aspects applied to them; and **Nanning** - through change in object implementation and new interfaces. The **JBoss** AOP framework supports addition and removal of aspect services independently of each other. These changes could be considered as planned if set of potentially useful aspects have been developed and provided with the applications, to be used as and when needed; or as evolutionary if these aspects have been developed later on, in accordance with changing requirements; or, in case of **AspectWerkz**, **Prose** and **JBoss**, if they need to be applied to the initially non-augmented pars of these systems. Evolutionary change requiring other types of change support, on the other hand, could require updating and extending the frameworks.

With respect to the frameworks themselves, JBoss is suited for evolutionary change due to use of Configuration paradigm in its design, where loosely coupled service components can be modified without affecting the rest of the system. The **AspectWerkz**, **Prose** and **Nanning** frameworks themselves have not been developed for change, though, due to use of OO for the their implementation, subclassing and interfaces could be used for static evolution.

The summary of discussion for this criterion is presented in Table 7.

Table 7: Summary for criterion 6: Programmed change robust to evolution.

AspectWerkz	JBoss	Prose	Nanning

[1] For instance, items from newly added modules of code, irrelevant to the initially defined pointcut, could be caught by it.

Supports addition, removal and updating of advice and introductions for applications, and more recently also addition of new aspects with their pointcuts. The framework itself is not designed for change.	AOP framework supports change in terms of use of aspectual interceptors. The server itself supports evolutionary as well as planned change.	Supports change through change of pointcuts and aspects in applications. Aspect Engine can be replaced, though the framework itself is not planned for dynamic change.	Supports change through proxies. The framework itself is not planned for evolutionary change.

5 Summary and Conclusions

In this paper we have discussed the problem addressed by the dynamic AO systems – dynamic reconfiguration with crosscutting concerns – and talked about mechanisms used to achieve it. We have provided a set of criteria for evaluating whether the problem is addressed effectively and discussed how several dynamic AO systems compare in this respect.

We have questioned how do the change implementation mechanisms affect the dynamic reconfigurability of systems where they are used in?

From the above discussion we can conclude that *hook weaving* (AspectWerkz, JBoss, Prose) allows for *loose coupling* of base and aspect code. When *actual* hook weaving is used (AspectWerkz, and possible option for JBoss and Prose) the reconfiguration *efficiency* of the systems is low for aspectising the none-augmented code (due to high *disturbance* and re-augmentation *time delay*) but the *application performance* is better, compared to *total* hook weaving (option in JBoss, Prose). The later option provides constant efficiency due to absence of non-augmented code, but poorer general application performance due to unnecessary processing from unused hooks. Total hook weaving also supports *programmed change robust to evolution* best as it has readily prepared hooks at possible joinpoints for any potential use. None of the systems uses *collected weaving*, which could be explained by the additional complexity required for dynamically unweaving the in-lined code.

The change implementation mechanisms do not seem to have a direct bearing on criteria such as *preservation of application integrity*, *application contribution to the reconfiguration process* and *reconfiguration specification.* Yet, these mechanisms could, of cause, be used to help address some issues (such as achieving *change synchronisation* by blocking execution in hooks for the whole duration of weaving process, as in Prose) in individual system implementations.

A notable trend in dynamic Java-based AO systems (including AspectWerkz, JBoss, Prose, as well as systems not discussed in this paper: e.g. JAC [20]) is that most of them turn to byte-code transformation rather then reflection. Although core Nanning Aspects does not resort to anything but reflection, frameworks based on top of core Nanning (such as cache, preveyler, etc.) do use byte-code manipulation. The likely cause of this is limited power of Java Reflection which supports introspection but not structural change (except for dynamic method hotswap in Java HotSpot VM).

These are our generic conclusions from the presented theoretical discussion. While a practical evaluation of the systems will clearly be beneficial we have postponed it to a future paper due to the lack of space. Nevertheless, it should be noted that an objective practical evaluation will have to be carried out against several scenarios, as each scenario could be better suited to a particular system's implementation. Moreover, we are aware that it could be difficult to make attributions due to the problems in identifying weather a specific result is achieved due to the underlying change mechanisms, or due to the implementation particularities of a given system.

References:

[1] J. Boner and A. Vasseur, "AspectWerkz Web Site, http://aspectwerkz.codehaus.org," 2004.

[2] J. Tirsen, J. Larsson, R. Lillsjo, J. Lind, and L. AB, "Nanning Aspects web pages , http://nanning.codehaus.org/overview.html AND http://nanning.snipsnap.org/space/Overview," 2003.

[3] G. Kniesel, P. Costanza, and M. Austermann, "JMangler - A Framework for Load-Time Transformation of Java Class Files," in *First IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, 2001.

[4] M. Fleury and F. Reverbel, "The JBoss Extensible Server," presented at International Middleware Conference (Middleware 2003), Rio de Janeiro, Brazil, 2003.

[5] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering, vol. 16*, pp. 1293-1306, 1990.

[6] I. Warren, "A Model for Dynamic Configuration which Preserves Application Integrity," in PhD Thesis, Computing Department. Lancaster: Lancaster University, 2000.

[7] C. Hofmeister, E. White, and J. Purtilo, "Surgeon: A packager for Dynamically Reconfigurable Applications," *IEE Software Engineering Journal, vol. 8*, pp. 95-101, 1993.

[8] J. M. Purtilo and C. R. Hofmeister, "Dynamic Reconfiguration of Distributed Programs," presented at 11th International Conference on Distributed Computing Systems, Texas, 1991.

[9] P. Oreizy, N. Medvidocic, and R. N. Taylor, "Architecture-Based Runtime Software Evolution," presented at International Conference on Software Engineering, Kyoto, Japan, 1998.

[10] J. Boner, "Self-defined aspects in AspecWerkz - new definition model http://blogs.codehaus.org/people/jboner/," 2003.

[11] A. Vasseur and J. Bonér, "AspectWerkz: A deeper view in the new “online mode” architecture, http://aspectwerkz.codehaus.org/downloads/papers/AspectWerkz-online-architecture.ppt.zip," 2003.

[12] B. Burke, A. Chau, M. Fleury, A. Brock, A. Godwin, and H. Gliebe, "JBoss Aspect Oriented Programming," The JBoss Group, http://www.jboss.org/developers/projects/jboss/aop, 2003.

[13] A. Popovici, G. Alonso, and T. Gross, "Just In Time Aspects: Efficient Dynamic Weaving for Java ." presented at 2nd International Conference on Aspect- Oriented Software Development, Boston, USA, 2003.

[14] A. Popovici, T. Gross, and G. Alonso, "Dynamic Weaving for Aspect-Oriented Programming," in *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, G. Kiczales, Ed., 2002, pp. 141-147.

[15] A. Popovici, A. Frei, and G.Alonso, "A proactive middleware platform for mobile computing," presented at 4th International Middleware Conference, Rio de Janeiro, Brazil, 2003.

[16] A. Popovici, G.Alonso, and T.Gross, "Spontaneous Container Services," presented at 17th European Conference for Object-Oriented Programming, Darmstadt, Germany, 2003.

[17] A. Popovici, G.Alonso, and T.Gross, "PROSE website http://prose.ethz.ch/Wiki.jsp?page=AboutProse," 2003.

[18] J. Boner and A. Vasseur, "AspectWerkz Source Code http://aspectwerkz.cvs.codehaus.org/viewcvs.cgi/aspectwerkz/?root=aspectwerkz," 2004.

[19] "Java Management Extensions (JMX) Overview, http://java.sun.com/products/jmx/overview.html," Sun Microsystems Inc., 2003.

[20] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java," presented at 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001.

Aspects in a Prototype-Based Environment.

Thomas Cleenewerck[1]
Kris Gybels[2]
Adriaan Peeters
Programming Technology Lab, Vrije Universiteit
{thomas.cleenewerck, kris.gybels, adriaan.peeters}@vub.ac.be

Abstract

Most of the existing aspect-oriented technologies are founded on a class-based object-oriented language. A whole other paradigm of object-oriented programming is thus ignored: prototype-based programming. In this paper we explore the impact of the differences of prototype-based and class-based programming on the design of crosscut languages and the implementation of weaving. Crosscut languages for prototype-based programs will definitely need to depend more on dynamic properties of the running program. In cases where some of those properties don't change too often, we propose the use of dynamically modifying code.

1 Introduction

Up to now Aspect-Oriented Programming seems to have been considered mostly in the context of class-based object-oriented languages. Class-based programs exhibit large amounts of static information that is exploited in many aspect-oriented systems. The static information about the structure of classes, their methods etc. gives crosscuts "something to hold onto" when trying to determine the points in the code they need to hook into. Weaving can be optimized by exploiting the static information, and for certain aspect languages can be done purely statically.

But besides class-based programming languages another more dynamic way of doing object-oriented programming exists: prototype-based programming. In this paper we describe a first exploration of what impact a prototype-based environment has on our notions of how aspects should be modeled, crosscuts be written and how weaving is to be done.

2 Prototype-Based OO

Prototype-based programming differs markedly from class-based object-oriented programming in that objects are not grouped into classes that specify the methods and data members all the objects in that group support. Rather each object is entirely individual and contains its own methods and data members. Examples of prototype-based languages are JavaScript, Pic%, Self [3] and Kevo [7].

This does not mean however that prototype-based programming does not support a form of "code reuse". While it obviously does not support class-based inheritance, it has a mechanism known as object-based inheritance or better "delegation". An object usually has a link to some

[1] This research is partially performed in the context of the e-VRT

Advanced Media project (funded by the Flemish Government) which consists of a joint collaboration between VRT, VUB, UG, and IMEC.

[2] Research assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

"parent" object. When a message is sent to the object, and the object does not have a method for the message, the message is automatically delegated to the parent. Note that delegation is different from mere forwarding in one important aspect: while it is handling the delegated request, any message the parent sends to itself is instead automatically sent to the child object [6]. Variations on this basic scheme are possible where for example objects can have multiple parents or a specific object to delegate to can be specified etc. It also may or may not be possible for the delegation link to be changed after the object is created etc.

Another difference with class-based programming is that objects are not created by instantiating a class. Rather they are constructed by cloning (or copying) an existing object. New or different methods and data fields can then be added to the new object.

3 Aspects with Delegation

Delegation is such a powerful mechanism it has been considered as a way of implementing certain aspects. The argument is that the concept of "putting code" before and after a method is well handled by delegation. Consider for example the situation depicted in Figure 1. It depicts a synchronized version of a point delegating to the actual point. The methods move, getX and getY of the synchronized point simply perform the necessary synchronization steps and further delegate the message to the parent point object. The synchronized point object can easily be reused by serving as a prototype for other synchronized points. When the synchronized point is cloned - depending on the semantics of the clone operation but this can usually be overridden - the parent point is also cloned and effectively a fully new synchronized point is delivered.

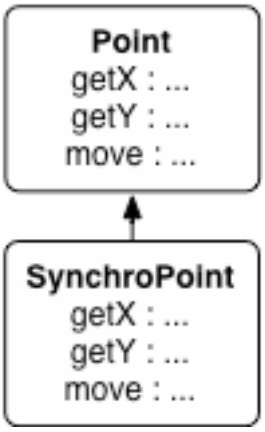

Figure 1: Point synchronization example with delegation.

It is however fairly obvious that using delegation to implement aspects suffers from some of the same problems as using the concept of wrappers in class-based languages to do the same. It is for example difficult to use the synchronization code for other objects than points. Furthermore a particular point object may have new methods added to it at run-time. One would have to remember to make sure an appropriate synchronized method is added as well.

So we can state that despite some claims to the contrary, the essential concept of low-level aspect-oriented programming may make sense for prototype-based programming as well. It would be useful to have objects that can describe using a crosscut when exactly they are to be invoked. The problem is then what should a crosscut language for a prototype-based language be able to express?

4 A Crosscut Language

Our view of a crosscut language is that it is a query language to select join points from the set of all join points that occur in a program [4]. In order to be able to be specific about which join points one intends, the crosscut language should not just allow one to express conditions on the join points themselves, but also on any of their associated objects like message arguments, or even the state of any other objects in the system. This is especially true for a crosscut language in a prototype-based environment. To see how exactly such an environment impacts the crosscut language we will start from the AspectJ crosscut language.

As is well known, the AspectJ crosscut language has primitive predicates for picking out join points based on their type, such as method calls, method executions, static initializers etc. It also has predicates for expressing conditions on the static extent of the join points, such as the class in which it is executed and from which class that class inherits. The join point type predicates also rely in a way on static code elements such as the types of arguments. As we will discuss, this is not possible in prototype-based languages.

4.1 Class

The most fundamental difference between prototype-based languages and class-based languages is the absence of classes. Let us describe the resulting repercussions.

- The most obvious repercussion is that static initialization is non existing in prototype-based languages.
- Classes group similar objects together. Reasoning about a class is reasoning about all its objects, and changing its class is changing all the objects. These are two ways in which aspect languages use classes. The reasoning is done in the pointcut language and the changing is performed in the weaving engine through advices. So aspect languages basically use classes to respectively *select* and *reach* the objects. In many of the AspectJ pointcut primitives, classes are used for this purpose. Consider the following pointcut: `call(* Point.setX(int))` which selects all the calls to the method `setX(...)` of class Point.
- Expressing this pointcut in a prototype-based environment is not so straightforward because there are only point objects available. Each instance containing its own set of methods and datamembers. Clearly we need some mechanism to reach or select the desired set of point objects.
- Classes are the blueprints of objects. In a class the methods that the object support are written, they are thus known in advance (at compilation time) and also determined in advance and never change. Reconsider the class Point, it contains one method `setX(...)` at compilation time. Because that never changes, all the objects of class Point will also be able to receive the method `setX(...)`. Therefore we can write the pointcut to select all the calls to the method setX(...) to all objects of class Point by the following pointcut expression `call(* Point.setX(int))`. The assumption in this pointcut is that in class-based languages we know in advance how all objects will behave.
- In prototype-based languages this is not the case. Each object is built ex-nihilo or by cloning a prototype object. Hence there is no static structure defining how the objects of a certain kind should look like. Each object may change the number and kind of messages it can receive and redefine its behavior. So objects are solely defined by their own set of messages they implement.

- Classes in the case of statically typed object-oriented languages are also used as a type. There are a number of primitive pointcuts that are especially there to select join points where the currently executing object or the target object is of a particular type respectively using `this(Point)` and `target(Point)`. Used in combination with the `call` primitive pointcut, the type of variables are used to determine which calls at the caller site are included in the joinpoint and which are not. Since the bulk of the prototype-based languages are dynamically typed, statically determining which calls to a certain object belong to a pointcut is not possible. Statically we may only rely on the signature of the method, checking if a variable is of a certain type must be performed at run-time.
- In class-based languages objects are created by instantiating a class. Aspect languages provide primitives to select join points of instance creation or object initialization `initialization(Point.new(int, int))` or `call(Point.new(int, int)`. Prototype-based languages however, need a very different way of creating objects since the lack of classes. Objects are either created ex-nihilo or are created by cloning another known object. In these cases it impossible to specify a joinpoint to intercept the creation of a new kind of object.

4.2 Inheritance

A second element of the static structure is inheritance. Similar to classes, inheritance hierarchies are also known and determined at compile-time and do not change during the execution of the program. The `within` primitive pointcut is one of the primitives that is based on that property. Because hierarchies do not change during the execution of a program, the hierarchies are a solid set and reliable set of information.

Since there are no classes in Prototype-based languages, it is not possible to define a static hierarchy based on the interrelationships of classes such as in Class-based languages. Another mechanism to specify interrelationships exists however: object inheritance. It can perfectly mimic class-based inheritance but it is much more flexible. Hierarchies can be formed, changed and broken at run-time. Moreover since these hierarchies are composed out of objects, it is also hard to refer to properties of these hierarchical structures.

4.3 Selecting Objects

As we have discussed in the previous section, crosscuts expressed in terms of the classical AspectJ pointcut primitives heavily rely on rather static structural information. In prototype-based languages the whole program structure is highly dynamic. Specifying pointcuts using these primitives is very difficult if not sometimes impossible because the only structural information that is left is the signature of messages. So the problem of specifying a point in prototype-based languages becomes actually the problem of selecting the desired objects.

Selecting the objects by merely the messages they support is often not sufficiently restrictive enough. Many popular messages are supported in many semantical totally different objects. Selecting objects using such messages would thus result in far too a heterogeneous set. For example, the following set of messages `{ add:, contain:, size: }` are part of many different kind of objects like sets, bags and even recordsets. To further narrow a search we must therefore incorporate state information. The state of an object determines the relationship the objects has with the rest of the objects and from the perspective of the object-oriented paradigm thus also contributes to the semantics of the object.

5 A Weaving Model

It seems we should at least try to make it possible to express conditions like "when a message send joinpoint occurs to an object whose parent supports the message m". The problem with such crosscuts is of course how to weave their associated advices efficiently, since the delegation relationships, the state of objects or even the messages they support can change dynamically in a prototype-based environment.

A problem with the crosscut language for a prototype-based environment is that almost none of the conditions can be resolved statically. In some prototype-based environments, delegation relationships can even change after an object was created. New slots can be added to objects for new data fields or even methods, meaning the object's type can even be changed. This implies weaving needs to be highly dynamic as well.

The most naive way of doing weaving when one is confronted with highly dynamic crosscuts is to do all weaving at run-time. As almost any execution of a statement in a program is usually considered to be a join point for weaving, this would imply that at *every* statement the dynamic weaver would have to go through the set of *all* crosscuts to see if any match with the current joinpoint and the current state of the program. This would obviously have an unacceptable performance impact.

5.1 Two-Phase Weaving

In previous work we and others already considered optimizing crosscuts that depend on dynamic properties of the application using techniques drawn from partial evaluation [4,5]. The idea is basically to split weaving into a dynamic and a static phase. In the static phase the set of crosscuts that need to be checked at a join point is reduced as much as possible. Because every join point is related to some statement in the code, information about the join points at that statement that can be statically derived from that statement can be used to check whether the crosscuts can *ever* match those join points. A simple example is when one has a crosscut that captures all message send join points where the message "test" is sent with a single argument greater than three. It is very simple to let a weaver figure out that a statement where the message "nottest" is sent can never lead to join points that match the crosscut. On the other hand, a statement where the message "test" is sent could lead to join points that match the crosscut, whether they actually do needs to be checked dynamically because the argument to the message is only known at run-time. When even more static information can be derived and is actually used in the crosscut, the number of crosscuts to check at each statement can be greatly reduced.

To actually implement the two-phase weaving model one can simply rely on some of the same techniques of purely static weaving. It is not necessary to really have an explicit dynamic weaver that intercepts every statement and checks the associated set of remaining potentially matching crosscuts. When most of those sets are empty, it is much better to simply wrap statements that have a non-empty set of potentially matching crosscuts. The code that would be wrapped around those statements specifies the set of crosscuts to be checked, and tells the dynamic weaver to check those. The weaver would then of course execute the necessary advices for the crosscuts that match.

5.2 Jumping Aspects Revisited

While generally applicable, the two-phase weaving model for turning a naive weaver into a more efficient one is too simple to handle most jumping aspects. The well-known jumping aspects problem refers to the problem that whether an aspect applies at a specific message send join point may depend on the calling context of the join point [2]. The `cflow` construct was added to AspectJ's crosscut language to deal with this problem. When used in a crosscut, a `cflow` specifies another (sub)crosscut and expresses that a join point needs to be preceded by another join point

on the call stack that matches the sub-crosscut. The straightforward way of implementing `cflow` in a naive weaver is to let it go through the call stack and check every preceding join point against the sub-crosscut until one is found that matches or the bottom join point is reached. Turning such a weaver into a two-phased one does not optimize this process, yet a pretty simple optimization is possible. Instead of going through the call stack to see whether a join point is to be found matching the sub-crosscut, one can set a flag when such a join point actually occurs and simply check the flag later on [5].

While the jumping aspects problem was recognized for control-flow like problems, it really applies to any kind of problem where the crosscut requires one to postpone weaving to the dynamic phase. In a sense the static location of such aspects also "jumps around" depending on conditions over the dynamic state of the program, whether it is the control-flow relationship between join points, the state of objects, their delegation relationships etc.

5.3 Blurring the Phase Distinction?

Simply postponing checks of such dynamic conditions by statically weaving if-conditions or the setting of flags into the code however may not be the only way to approach this problem. As we've already mentioned, an interesting property of most prototype-based systems is that they allow for easy replacement of methods at run-time. This would allow us to blur the strict separation of having a static weaving phase where code is modified and a dynamic phase where some of the dynamic conditions in the added code are checked. Instead, code could also be changed at runtime as a way of making the static location of aspects "jump around", quite literally in fact.

The way this would work for a `cflow` example is as follows. Recall the cflow is a condition of some crosscut, and the cflow itself specifies a sub-crosscut. As described earlier this can be woven with code that sets a flag when a join point is encountered that matches the sub-crosscut, and code that would check that flag at the static locations of join points that might match the whole crosscut (as well as any other dynamic conditions of that crosscut). We could change this to work like this: instead of setting a flag when the sub-crosscut is matched, we could change at *run-time* the code where we would need to check for the match to the whole crosscut as before, minus the check for the flag.

While this approach of literally making code jump around is quite a bit of overkill for something like a `cflow` it may be interesting to consider for other kinds of dynamic conditions. The problem with the `cflow` is that it would probably require too many code jumps as any cflow condition is one that quite often switches between being true and false. However, other kinds of dynamic conditions, such as for example delegation relationships between objects in prototype-based programming may not change that often. Let us consider a crosscut that would express something like "all message send join points of the message M to an object whose delegation parent is X" as a rather abstract but useful example. In this case it might be more interesting to weave this by changing the message M of any object that has X as parent, as well as changing this weaves dynamically when X gains or loses new delegation children, than to weave a check at every method M because any object might have X as parent at some point.

6 Position

While we realize our discussion of aspects in prototype-based environments is very preliminary, we do feel we have valuable contributions to make to the workshop's topic of discussion:

- We offer a "fresh" perspective on the relationships between aspects, objects and dynamic joinpoint models by considering the more dynamic variant of the object-oriented paradigm, prototype-based programming. Taking this paradigm as a starting point for

further exploration may lead us to automatically consider more dynamic features for crosscut languages.

- We are interested in discussing what exactly the impact is of not having classes or types for crosscuts to grab onto will do to how we specify crosscuts. Will they naturally need to be more dynamic or can anyone offer a counter-view? Has aspect-oriented programming not been too depending so far on having static, rigid lexical structures available to specify crosscuts? One potential area of investigation is the use of query languages drawn from the field of object-oriented database systems [1]. Another potential area is to use inductive logic programming techniques to classify objects.
- Is the idea of literally making "jumping aspects" jump around by doing dynamic code changes for aspects that don't require jumping too much but would otherwise require a lot of dynamic checks a worthwhile technique to further explore? What support would such weaving exactly require from the run-time environment?

Acknowledgments

We would like to thank Jessie Dedecker, Johan Fabry and Dirk Deridder for the interesting discussions and insights they provided us.

References

[1] Query by example, http://www.db4o.com/.

[2] Johan Brichau, Wolfgang De Meuter, and Kris De Volder. Jumping aspects. In Peri Tarr, Maja D'Hondt, Christina Lopes, and Lodewijk Bergmans, editors, International Workshop on Aspects and Dimensional Computing at ECOOP, 2000.

[3] Randall B. Smith David Ungar. Self: The power of simplicity. LISP AND SYMBOLIC COMPUTATION, 1991.

[4] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In Proceedings of the Second International Conference of Aspect-Oriented Software Development, 2003.

[5] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In Gary T. Leavens and Ron Cytron, editors, Foundations of Aspect-Oriented Languages Workshop at AOSD 2002, number 02-06 in Tech Report, pages 17–26. Department of Computer Science, Iowa State University, 2002.

[6] Klaus Ostermann and Mira Mezini. Object-oriented composition untangled. In Proceedings OOPSLA '01, Tampa Bay, FL, 2001.

[7] Antero Taivalsaari. A Critical View of Inheritance and Reusability in Object-oriented Programming. PhD thesis, University of Jyväskylä, 1993.

Dynamic Aspect-Oriented Programming: an Interpreted Approach

Fabrício de Alexandria Fernandes
Thais Batista
Informatics Department
Federal University of Rio Grande do Norte
Natal – RN – Brazil
fabricio@ppgsc.ufrn.br, thais@ufrnet.br

Abstract

In this position paper we argue that using an interpreted and dynamically typed language – Lua – it is possible to integrate components and aspects at runtime. We describe AspectLua – a Lua extension to support aspect-oriented programming. AspectLua allows the dynamic definition of aspects and it also offers support for dynamic weaving. The difference of AspectLua to others aspect-oriented languages is the fact of being interpreted and dynamically typed – this allows the insertion/removal of components and aspects in an application at runtime.

1 Introduction

Component-based software development is a current trend in software engineering because it promotes the idea of reusing components. It focuses on composing applications by assembling prefabricated piece of software named components [2]. In this context, the development process is split in two levels: at the *component level* there are the components that offer some services. At the *configuration level*, an application is specified through a program that contains the components that provide the application functionality, as well as the interconnection between the components.

Following the idea of separation of concerns, configuration-based programming emphasizes the need of decoupling concerns related to coding computational components from those relating to binding components and creating a new application. Thus, configuration based programming separates the composition between components from the component implementation that makes part of this composition.

Another research area that has gaining popularity as a promising approach to software engineering is Aspect-Oriented Programming (AOP) [1] because it emphasizes separation of concerns and promotes the benefits of modularity. By decoupling concerns related to coding computational components from those relating to non-functional aspects, AOP improves reusability of software.

In this position paper we address the integration of component-based software development and aspect-oriented programming by discussing how non-functional aspects of an application can be dynamically integrated with the components at the configuration level. In order to handle this integration, we adopt an interpreted-based approach where the aspect weaving occurs at runtime.

We aim to handle application dynamic reconfiguration where it is possible to define and redefine aspects at runtime as well as to insert and to remove components in the configuration.

In general, aspect-oriented approaches are static – aspect code and functional modules are mixed at compile time (static weaving). Besides, a special compile is needed to combine the aspect code with the base code. Although this strategy avoids type mismatches, it imposes many restrictions on application evolution. In contrast, in this work we use a dynamic approach, where crosscutting concerns can be inserted and removed from an application configuration at runtime.

In this work we present AspectLua – an extension of an interpreted and procedural language (Lua [3]) to handle aspect-oriented programming. AspectLua allows the dynamic definition of aspects and it also offers support for dynamic weaving. Components and aspects are combined at the configuration level. We choose the Lua language because it is dynamically typed and because it provides facilities for extending its behavior without modification in the underlying interpreter. Such facilities are explored in the definition of AspectLua. We argue that this introduces a different style for aspect-oriented programming where dynamism is a key issue, weaving is done at runtime and both components and aspects can be inserted and removed from the application at runtime.

This paper is organized as follows. Section 2 presents the Lua language and the basic concepts of aspect-oriented programming. Section 3 presents AspectLua and discusses its support for defining applications by combining components (the functional code) with aspects. Finally, section 4 contains the final remarks.

2 Basic Concepts

2.1 Lua Language

Lua is an interpreted extension language developed at PUC-Rio [3]. Lua is dynamically typed: variables are not bound to types although each value has an associated type. Lua includes conventional features, such as syntax and control structures similar to those of Pascal, and also has several non-conventional features, such as the following:

- Functions are *first-class* values, which means they can be stored in variables, passed as arguments to functions, and returned as results. Functions may return several values, eliminating the need for passing parameters by reference.
- Lua *tables* implement associative arrays, and are the main data structuring facility in Lua. Tables are dynamically created objects and can be indexed by any value in the language, except nil. Many common data structures, such as lists and sets, can be trivially implemented with tables. Tables may grow dynamically, as needed, and are garbage collected.
- Lua offers several reflexive facilities. One simple example is the *type* function, which allows a program to determine the type of a value. Metatables are Lua's most generic mechanism for reflection. Several situations in which the interpreter would intuitively generate an error can be captured by a programmer-defined function, called a metatable. Examples of such situations are calls to non-existent functions and indexing a value that is not a table.

2.2 Aspect-oriented programming

Aspect-Oriented Programming emphasizes the need to decouple concerns related to coding computational components from those relating to non-functional aspects of an application. This decoupling is often supported by proposing the use of different languages for programming these two types of activities [5]. For instance, Java programmers write the functional code in Java and

the aspect code in a Java extension for aspect-oriented programming such as AspectJ [1]. A special compile does the integration between Java and AspectJ programs.

Although there is no consensus about the terminology of the elements that makes part of aspect-oriented programming, we refer in this work the terminology used in AspectJ because it is the most traditional aspect-oriented language. *Aspects* are the elements designed to encapsulate crosscutting concerns and take them out of the functional code. *Join Points* define the composition of base components and aspects. *Advices* define codes that run at join points. They can run at the moment a joint point is reached and before the method begins running (before), at the moment control returns (after) and when the joint point is reached (around).

3 AspectLua

Lua does not offer support for aspect-oriented programming. In order to include this issue, AspectLua extends the language by including support for aspect programming. It is being developed exploring the Lua extensions facilities. Due to the Lua reflexive features it is not necessary to modify the Lua interpreter to support AspectLua. AspectLua includes elements for the definition of aspects, pointcuts, join points and advices. In the current implementation it only supports method-intercepting calls.

3.1 Architecture

The development of applications using AspectLua follows the idea presented in Figure 1.

- The **component level** contains the application base code (the components). They implement the main functionality of the application regardless of the non-functional issues.
- The **aspects definition level** contains the non-functional aspects of the application. It is defined using the AspectLua constructions.
- The **application configuration level** specifies the components and the aspects that compose the application.

The two codes (components and aspects) are executed by the Lua interpreter that invokes AspectLua when executing commands associated with aspect programming. This is done transparently for the programmer and supported by the reflexive facilities offered by Lua. In contrast to the traditional aspect-oriented programming, in this approach there is no need of a special compile to join aspect with the functional modules, the interpreter at runtime does this mixing.

3.2 AspectLua Elements

Simplicity and flexibility are the main features of the Lua language. AspectLua follows the same idea and preserves these features. No special commands are needed. It offers only two Lua instructions to the definition of aspects, pointcuts, join points and advices. AspectLua defines an *Aspect* object that handles all aspects issues. This object defines a function to create a new aspect: the *new()* function.

After creating a new aspect, it is necessary to define a Lua table that contains the aspect elements (name, pointcuts and advices). Figure 2 illustrates the generic code to aspect definition. The first parameter is the aspect name. The second parameter is a Lua table that defines the pointcut elements: its name, its designator and the function that must be intercepted. The designator defines the pointcut type (the current implementation of AspectLua only allows method-call-interception). The third parameter is a Lua table that defines the advice elements: its type (after, before, around) and the action to be taken when reaching the pointcut.

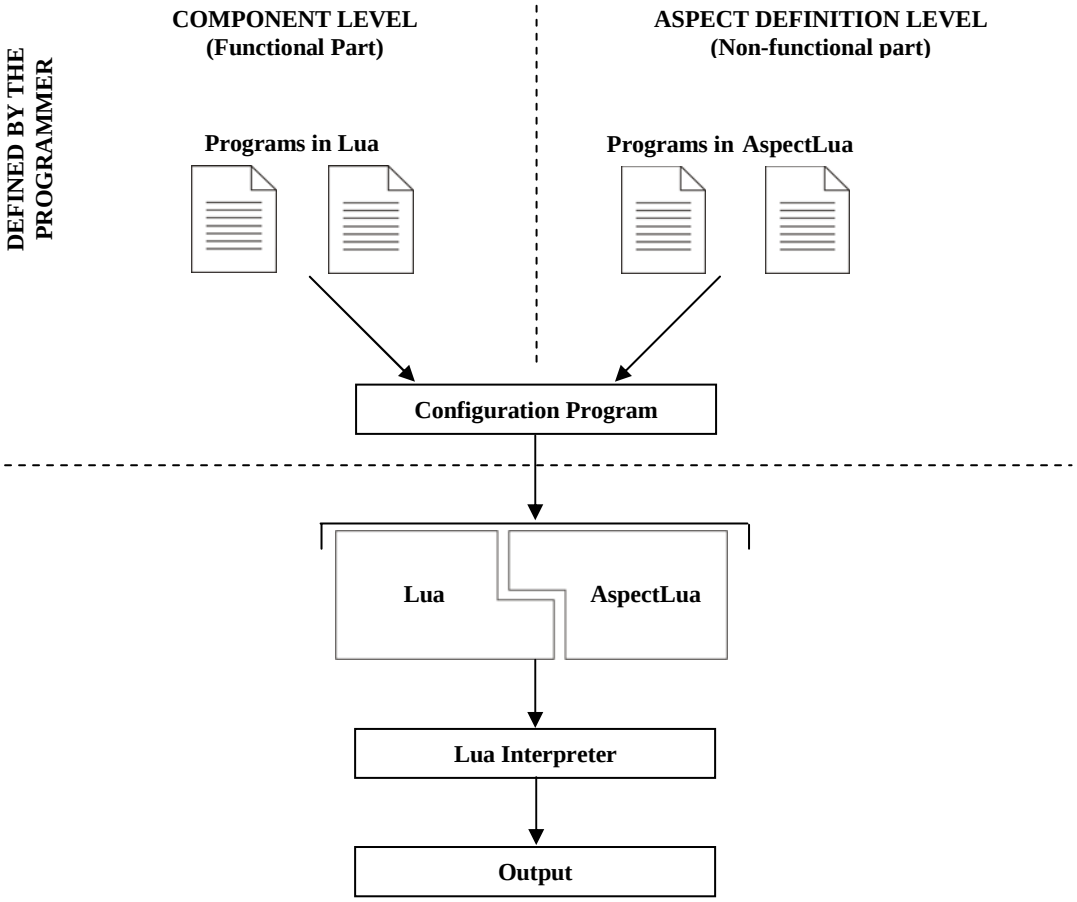

Figure 1. Architecture.

```
a = Aspect:new()
a:aspect({name = 'value'},
         {pointcutname = 'value',
          designator = ' value ',
          function = value},
          {type =' value', action = value})
```

Figure 2. Generic code to aspect definition.

3.3 Example

To illustrate the use of AspectLua, a simple example was developed. The main purpose of the application is to query a database. As an additional feature, we define functions to log all database access. Figure 3 shows the code of application mixed with the logging code.

```
-- Application example.

function queryDB()

-- Function code for querying a register in a database

-- Function call for logging
logging()
end
```

```
function insertDB()

-- Function code for inserting a register in a database

-- Function call for logging
logging()
end

function removeDB()

-- Function code for removing a register in a database

-- Function call for logging
logging()
end

function logging()
     print('DB accessed at: ' .. os.date())
end

queryDB()
removeDB()
```

Figure 3. Lua application code.

Exploring the AspectLua support, the code can be separated and the additional function can be implemented as an aspect. Figure 4 illustrates the functional code and Figure 5 shows the aspect code.

```
-- Application example used to illustrate the AOP.

function queryDB()
-- Function code for querying a register in a database
end

function insertDB()
-- Function code for inserting a register in a database
end

function removeDB()
-- Function code for removing a register in a database
end
```

Figure 4. Application functional code.

```
-- Code for create an Aspect.
-- A definition of an aspect is based on the code described in
aspect.lua
-- The functions represent the code for the advice.

-- Function that logs the access to the database
function logging()
        print('DB accessed at: ' .. os.date())
end

-- Aspect Definition
a = Aspect:new()
a:aspect({name = 'SimpleLogging'},
```

```
        {name = 'loggingFunctions', designator = 'call',
functionName = queryDB},
        {type ='after', action = logging})
```

Figure 5. Aspect code exploring AspectLua.

The code of Figure 5 creates an aspect named *SimpleLogging*, with a pointcut named *loggingFunctions* that determines the *method-call* interception of the *queryDB* function. The *logging* function must be executed *after* the invocation of queryDB.

Since functions are first-class values in Lua, they can be passed as a parameter of a Lua table. This introduces a great flexibility in defining the actions to be taken by aspects.

Although we adopted the traditional linguistic structure of aspect-oriented programming, this is quite different using Lua. It is not necessary to introduce special commands in AspectLua to express the aspect elements, we explore the Lua data structuring facility (tables). Since Lua is interpreted and dynamically typed, the aspect elements are not static, they can be easily inserted, removed and modified via the Lua interactive console.

A Lua configuration file define the files that contain the application base code and the aspect code. Figure 6 shows the configuration file that defines the DB application file and the Log aspect file. Again, due to the dynamic style of Lua, the configuration program can also contain dynamic decisions. For instance, a different aspect code can be used according to conditions expressed at the configuration file and verified at runtime. This adds a great deal of flexibility to the configuration.

```
dofile("queryapplication.lua")
dofile("simplelogginaspect.lua")
```

Figure 6. Configuration File.

Finally, the Lua interpreter receives the configuration file and executes the application producing the output with the logs. This is illustrated at Figure 7.

```
DB accessed at: 2004/01/11 16:40:20
```

Figure 7. Output.

4 Final Remarks

In this paper we proposed the use of an interpreted language and dynamically typed language to support dynamic aspect weaving. Aspects are defined using AspectLua – a simple extension of Lua – and the integration of components and aspects are defined at a configuration level via a Lua program. The reflexive features of Lua allows that the Lua interpreter combine the two parts of the application. The interpreted approach makes possible to define aspects and components at runtime.

The flexibility of Lua introduces a different style to aspect-oriented programming where it is not necessary to use a different language to define aspects. Besides, it is not necessary to implement a special compile or interpreter to mixing the components with the aspects. This interpreted approach allows application-specific customization at runtime. A similar support for aspect-oriented programming would be applicable using other interpreted language. However, the language must have features similar to those of Lua such as dynamic typing and extensions facilities.

As a future work we intend to investigate the integration of this aspect-oriented approach with the CORBA component model where the application is composed by CORBA components (the functional code) and aspects.

References

[1] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr and H. Ossher. Discussing Aspects of AOP. Communications of the ACM, Vol. 44, No. 10, pp 33-38, October 2001.

[2] C. Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998.

[3] R. Ierusalimsky, L. H. Figueiredo and W. Celes. Lua – an extensible extension language. Software: Practice and Experience, 26(6):635-652. 1996.

[4] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa. Abstracting Object-Interactions Using Composition-Filters. Object-Based Distributed Processing. R. Guerraoui, O. Nierstrasz and M. Riveill (eds), LNCS, Springer-Verlag, pp 152-184, 1993.

[5] G. Papadoupoulos and F. Arbab. Coordination Languages and Models. In Advances in Computers. Academic Press. 1998.

A Concern-based Approach to Dynamic Software Evolution

Peter Ebraert [1]
Eric Tanter [2,3]
[1] Vrije Universiteit Brussel, PROG Pleinlaan 2, Brussel Belgium
[2] University of Chile, DCC/CWR Avenida Blanco Encalada 2120, Santiago, Chile
[3] Ecole des Mines de Nantes, 4, Rue Alfred Kastler, Nantes, France
pebraert@vub.ac.be, etanter@dcc.uchile.cl

Abstract

The problem we are trying to tackle in this research is that of the availability of critical applications while they are being updated. We are investigating how dynamic aspects can be used to evolve some concerns of a running application. Our approach targets a three steps process: first, an application is statically refactored so that evolvable concerns are cleanly separated, second, concerns are handled by a reflective infrastructure, and third, a runtime API to this infrastructure makes it possible to update the separated concerns one at a time.

1 Introduction

An intrinsical property of a successful software application is its need for evolution. In order to keep an application up to date, we continuously need to adapt it. Usually, applications have to be shut down before they can updated, in order to avoid data corruption for example, but mostly because it is generally not possible to update an application at runtime. In some cases, this is beyond the pale, for example in critical systems such as web services, telecommunication switches, banking systems, etc. Unavailability of software systems could have unacceptable financial consequences for the companies and their position in the market.

Currently this problem is being solved using redundant systems [1]. Every critical system is provided with a double that can take over all the functions of the original one, whenever it is not available. This solution has already proved it is working well but it still has some disadvantages. Firstly, the financial side of the story: every piece of hardware and software has to be purchased multiple times. Also, redundant systems have their own problems as the management of the different versions gets harder. Also the maintenance and the switching between the redundant systems are sometimes underestimated.

We are investigating a better, more flexible, solution to this problem, based on the development of applications with separated concerns [2]. In such an application, every addressed concern -- crosscutting or not -- exists as a separate entity that can be adapted and substituted without affecting the rest of the system. What we call an *entity* in this paper is a part of the program that copes with a certain concern. Depending on which programming paradigm is used, the entity can be of different types. In AOP for example, an entity refers to a set of objects that handles one function or aspect, but an entity can also refer to a component, a set of functions, a set of procedures, a set of abstract data types, etc., depending on the programming paradigm.

If the application is cleanly split up in separate entities, we can say that the evolution of that software system falls down to the removal, the addition or the modification of a certain separated system entity. From the moment that this can be done while the application is running, we can talk about *dynamic software evolution*.

In real life, it turns out that the principle of separated concerns is not always that easy to achieve. Even though some experimental techniques already exist to that matter, they are not exerted all the way through. The major part of today's applications is only a set of strongly woven concerns. Next to that, most existing techniques in separation of concerns are still too static to support dynamic maintenance in real time, because they provide a model in which concerns are fixed in the application [3-6]. This hinders their modification, at execution time. More dynamic techniques are to be investigated to solve that issue. Several prototypes of those techniques do exist [7, 8], but still lack some dynamic properties as well as practical experience.

2 A Concern-based Approach to Dynamic Software Evolution

In order to solve the problems stated above, there are two fundamental issues we have to cope with. On the one hand, we should make sure that the systems match the principle of separated concerns. On the other hand, we should find a technique that allows cleanly developed systems to evolve dynamically.

2.1 General approach

The opening perspective of our research is that we will allow all kinds of entities to evolve dynamically. This is where our approach differs from existing approaches, for instance the ones based on the Sun's JVM hotswap API. While those will only allow the change of class implementations, we aim to allow the change of entity implementations. This is a very ambitious perspective, so we will try to get as close to it as possible by using a step by step approach.

As a start, we will test this approach on the separated aspects of an aspect-oriented application. In such an application, evolvable concerns (aspects) are intrinsically encapsulated out of the base application (this is the obliviousness property). This makes it a lot easier to consider their evolution, compared to evolving pieces of the rest of the base program, which will always be interacting with the other modules.

In a second phase, we want to evolve cleanly separated entities of an ordinary object oriented application which are still well modularized, but not implemented as an aspect. This will be a harder challenge as the application has direct knowledge and references to such modules.

As the ultimate goal is to allow the same approach for functional, procedural, and programming paradigms, we should finally widen our field of action.

2.2 Matching the principle of separated concerns

Most of existing applications do not match with the principle of separated concerns. Nevertheless, they should match with it, if we want to allow them to evolve dynamically. For that, we want to investigate how aspects can be detected in existing applications. Research in that domain, *aspect mining*, only started recently. Although abstract results seem promising, concrete results are not yet available.

We need to investigate if and how we can detect different concerns by observing the dynamic behavior of the application and by deriving which parts are weakly or strongly connected, which communication patterns occur frequently, etc. We plan to use *reflection* -- calculations on calculations -- in order to do so. In [9] a reflection based runtime monitor is presented, which is able to observe a running application. Extending that monitor with some domain knowledge on communication patterns would already allow the detection of some concerns. The use of a logic metalanguage is also an envisioned possibility.

Once we have identified the different concerns, we need to restructure the application in order to make the concerns explicit. For this, we use refactoring techniques [10], which permit the modification of the internal structure of an application without influencing its semantics. We may need to extend existing refactoring techniques for our purpose.

2.3 From runtime reflection to dynamic evolution

A reflective system is able to reason about itself by the use of metacomputations -- computations about computations. For permitting that, such a system is composed out of two levels: the base level, housing the base computations and the metalevel, housing the metacomputations. Both levels are said to be causally connected. This means that, from the base level point of view, the application has access to its representation at the metalevel and that, from the metalevel point of view, a change of the representation will affect ulterior base computations. Depending on which part of the representation is accessed, the part describing the structure of the program, or the part describing its behavior, reflection is said to be structural or behavioral.

Figure 1 illustrates the causal connection between base and metalevel, and shows how this can be used in order to change the behavior or the structure of a base-level application. The left part of the figure shows the architecture of a certain application that has cleanly separated entities at the base level. The metalevel houses a representation of this application. Using dynamic structural and behavioral intercession, the application could self-evolve through metalevel manipulations. The center picture shows that a new entity is added in the metalevel representation of the application. The right picture shows the propagation of the metalevel change down to the base level, thus changing the application's behavior and structure. Using this approach we can update separated entities of a system without having to switch off the system, and thus allow dynamic evolution. Still there are several issues that have to be solved in order to do so.

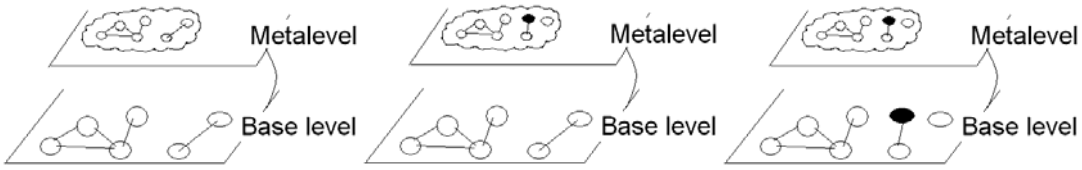

Figure 1. Dynamically updating an entity through metalevel manipulation.

2.4 The evolution framework

We need a platform that provides both structural and behavioral reflection at runtime and that allows dynamic composition of meta-entities. As a first step, such entities will be aspects. Since we need structural reflection at runtime, we are going to experiment with Smalltalk. The behavioral reflection part will have to be added, based on the ideas of partial behavioral reflection as exposed in [10] and materialized in the Reflex platform for Java. Finally, we plan to inspire from the work on EAOP (Event-based Aspect-Oriented Programming) [11] with regards to dynamic aspect composition facilities. Although targeted to behavioral issues, Reflex and EAOP underlying ideas can be adapted to deal with structural changes. First, we definitely retain the idea of a global monitor controlling the application, and the selective introduction of hooks within base applications. As long as structural changes are intra-entity -- stay locally inside a certain entity -- they are straightforward to allow. If they are inter-entity changes, things will obviously get more complicated as we will have to keep track of the inter-entity dependencies. This is an issue that we will have to investigate further.

In a first version of the framework, we plan to apply a two-layered architecture to allow us to modify the behavior of a running application even when it is already running. For doing that, we instrument the running application with calls to the monitor at every point where communication between entities occurs. The monitor has to keep track of that communication in order to make it possible to substitute a certain entity. During execution, the monitor passes control to the concerned entities, making its presence unnoticeable. When changing a given entity, the monitor will queue all calls to the 'old' entity in order to send them to the 'new' one once in place. This is

illustrated in Figure 2. Our approach implies that any evolvable entity has to be referenced by the monitor, and that the monitor keeps track of entities and inter-entity relations. The results of the number of iterations for various values of the tombstone problem are shown in Table 1. It remains an intriguing open problem as to whether the tombstone function terminates for all positive integers.

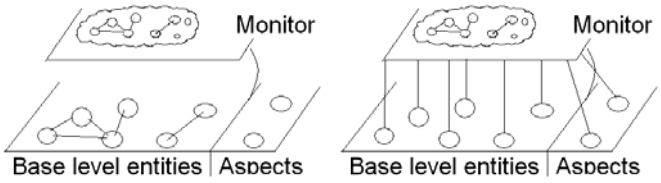

Figure 2. Runtime evolvability by means of a two layered architecture: inter-entity communications (left) are indirected to the monitor (right).

2.5 The runtime API

Finally, in order to evolve the application, the user has to change the application's representation in the monitor. To that extent, a runtime API will be included so that the user can interact on-line with the monitor. The functionalities of the API have to include the addition, the removal and the modification of a system entity (aspect or functionality). Adding a new entity is done by writing its code, and by registering it in the monitor. Removing an entity is more complex, as we should make sure that no other entities are dependant of that entity before actually removing it. If that is however the case, the programmer should be warned about that. When a certain entity needs to be modified, we have to write the new entities code, and tell the monitor that it should use the new entity instead of the old one whenever the old one is referenced by an other system entity. In this case, there are also some difficulties that arise, since we should be able to transfer the state from one to another entity. Some formal definition of the before-after behavior should be established in order to avoid conflicts.

3 Conclusion

In this paper, we started to investigate the issue of dynamic evolution of applications. We have sketched a three-step process based on cleanly separating evolvable concerns in an application, controlled at the metalevel by a monitor with full reflective capabilities. Such a monitor merges the ideas of EAOP and partial behavioral reflection with the great dynamic capabilities of a language like Smalltalk, to provide dynamic evolution of object-oriented and aspect-oriented applications.

References

[1] O' Connor, P.: Practical Reliability Engineering. 4th edition. Wiley (2002)

[2] Dijkstra, E.: The structure of THE multiprogramming system. Communications of the ACM 11 (1968) 341-346

[3] Ossher, H., Tarr, P.: Hyper/J: multi-dimensional separation of concerns for java. In: Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland (2000) 734-737

[4] Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (1998)

[5] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Proceedings of the European Conference on Object-Oriented Programming, Lecture Notes in Computer Science,. Volume 2072, Springer Verlag (2001) 327-353 http://aspectj.org

[6] Aksit, M., Tekinerdogan, B.: Aspect-oriented programming using composition filters. In Demeyer, S., Bosch, J., eds.: Object-Oriented Technology, ECOOP'98 Workshop Reader, Springer Verlag (1998) 435

[7] Popovici, A., Alonso, G., Gross, T.: Just-in-time aspects: efficient dynamic weaving for java. In: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press (2003) 100-109

[8] Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In Batory, D., Consel, C., Taha, W., eds.: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002). Volume 2487 of Lecture Notes in Computer Science., Pittsburgh, PA, USA, Springer-Verlag (2002) 173-188

[9] Tanter, E., Ebraert, P.: A portable yet flexible approach to interactive runtime inspection. In: ECOOP Workshop on Advancing the State-of-the-Art in Runtime Inspection (ASARTI 2003). (2003) Darmstadt, Germany.

[10] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)

[11] Tanter, E., Noye, J., Caromel, D., Cointe, P.: Partial behavioral reflection: Spatial and temporal selection of reification. In Crocker, R., Steele, Jr., G.L., eds.: Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003), Anaheim, California, USA, ACM Press (2003) 27-46 ACM SIGPLAN Notices, 38(11).

Garbage Collection in Jikes: Could Dynamic Aspects add Value?

Celina Gibbs
Yvonne Coady
University of Victoria
{celinag, ycoady}@uvic.ca

Abstract

This paper provides an overview of some of the specific structural relationships apparent within garbage collection in the Jikes Research Java Machine. The preliminary results show the structure of one aspect of garbage collection, and highlight the advantages run-time configuration options could have with respect to garbage collection in general.

1 Introduction

The Jikes Research Virtual Machine (RVM) [1] affords researchers the opportunity to experiment with a variety of design alternatives in virtual machine infrastructure. The project is open source and written in Java. One of the core system elements that stands to receive much attention in this testbed environment is garbage collection (GC). State of the art technologies for improving GC performance are still evolving, in particular in multiprocessor systems.

The benefits of GC are well known and have been appreciated for many years in many programming languages. GC separates memory management issues from program design, in turn increasing reliability and eliminating memory management errors. Though GCs have improved significantly over the last 10 years, ongoing research aims to further reduce costs and meet application specific demands. Costs not only include performance impact, but also configuration complexity. In 1.4.1 JDK, for example, there are six collection strategies and over a dozen command line options for tuning GC [3]. Basic strategies, such as reference counting, mark and sweep, and copying between semi-spaces have been augmented with hybrid strategies such as generational collectors. These are collectors that treat different areas of the heap with different collection algorithms. Collectors not only differ in the way they identify and reclaim unreachable objects but, they can also significantly differ in the ways they interact with user applications and the scheduler.

Currently, the Jikes RVM comes with eight different strategies, known as *plans*. Advice for developers who want to add a new plan can be found in the Jikes User Guide [2]:

> *"A good way to start is to compare some of the different plans and understand the significance of the differences."*

Our experiment focuses on one element of garbage collection that crosscuts all plans – the memory allocation policy. Using a standard configuration for Jikes, we refactored this concern using static aspects as a first step. We argue that, in this form, the significant differences between each plan's use of policy becomes clear, and the internal structure of policy can be represented as a simple finite state machine. Developers could then more easily identify the differences, and hence leverage this structure to understand their significance, and develop new plans. We further

propose that run-time configuration would enable a wider range of development opportunities, such as feedback-based optimization.

This paper proceeds as follows: Section 2 describes what a developer is faced with when analyzing the relationship between plan and policy based on manual inspection of the packages involved. Section 3 presents this same relationship based on a perspective that includes control flow information. Section 4 presents an implementation of the policy/plan relationship using a static aspect, and describes why we believe dynamic aspects could add value to garbage collection developers working with Jikes.

2 Plan and Policy: Manual Inspection

The two packages of interest in our experiment are plan and policy. Each plan details a particular configuration of the components involved in GC, where policy is one such component. These components then together define the memory management strategy for a particular build of the Jikes RVM.

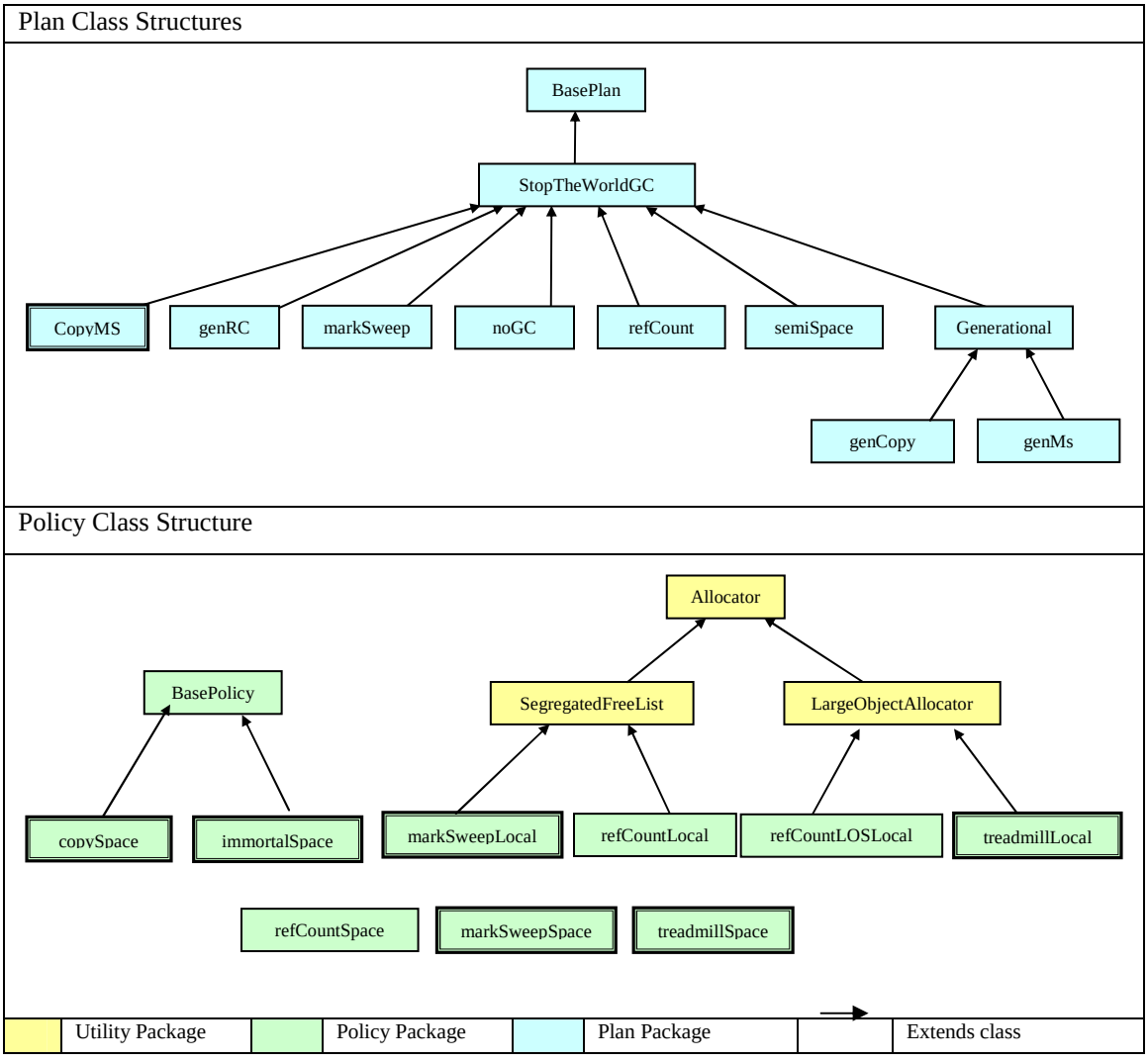

Figure 1. Class Structures

The classes involved in plan and policy have separate and independent structural hierarchies, as illustrated in Figure 1. In more detail, the top half of Figure 1 shows that all current plans inherit from the `BasePlan` class. The bottom half of Figure 1 illustrates that some policies inherit from the `BasePolicy` class in the policy package, and some inherit from classes in the utility package, while some policies do not extend any class.

In the plan package the `BasePlan` holds the frame work for all memory management schemes. It is extended by `StopTheWorldGC` which implements the base functionality for the stop-the-world collector plans. Stop-the-world collector plans are those that require the user program to be suspended while collection takes place. Classes touched by our experiment are outlined with a double lined border: the *CopyMS* plan, which in turn uses policies *copySpace*, *ImmortalSpace*, *markSweepSpace/Local*, *treadmillSpace*/*Local*.

3 Plan and Policy: A Protocol-Based Perspective

All plan classes divide *global* and *thread-local* activities. Global activities require synchronization between collector threads on different processors in a multiprocessor environment, while thread-local activities do not[1]. Figure 2 highlights four of the key methods in the `StopTheWorldGC` class where plans interact with policies: *globalPrepare()*, *threadLocalPrepare()*, *threadLocalRelease()* and *globalRelease()*.

The large arrow in Figure 2 shows that the program executes each of the four paths exclusively and sequentially. This ordering is a critical part of GC protocol correctness. Each one of these four control flow paths leads to a collector determined/defined plan class. Each class that extends `StopTheWorldGC` has its own unique version of the four synchronized methods, *globalPrepare()*, *threadLocalPrepare()*, *threadLocalRelease()* and *globalRelease()*.

In each of these four methods, invocations to methods within policy classes are made. This shows how policy is staged throughout the control flow of plan. For each plan, each of these four points along the execution path invoke their own particular combination of policy methods. As a result, to understand the significance of the differences between how each plan is implemented, a developer must understand when and which policy objects are used along this staged execution path through plan.

[1] We have not analyzed the effects of synchronization yet, but plan to investigate the mater further in the continuation of this work.

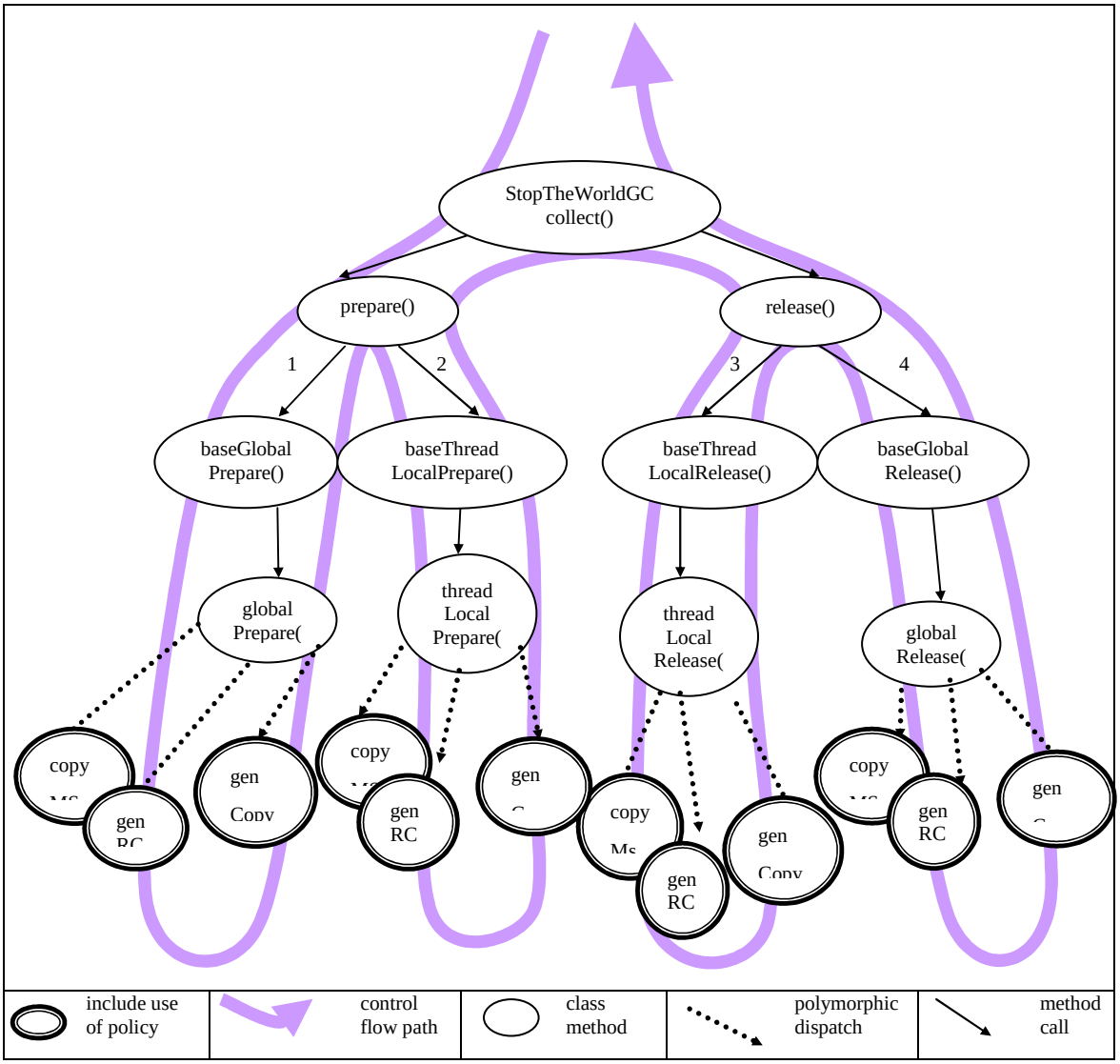

Figure 2. Garbage Collection Control Flow Map

4 Understanding the Differences

Figure 3 overviews the high-level structural relationships between plans and policies. The solid arrows illustrate the relationship between the various plans and their associated policies. Additionally, the internal structure of policy is represented by the separation of the policy classes with respect to the control flow. The two natural groups that are formed are those used by the global paths, aligned along the left of the frame and those used by the local paths, aligned along the right of the frame. As the figure shows, the global paths employ only **Space* classes, whereas the local paths employ only **Local* classes.

The group of *Space* classes can further be split into two groups. The first is the group of three classes that do not inherit from any other classes. There is a relationship between these *Space* classes and the corresponding *Local* classes with similar names. Additionally, there is an internal structure to policy, where each of the *Local* classes take an instance of the associated *Space* class as a parameter, as indicated by the dashed arrows.

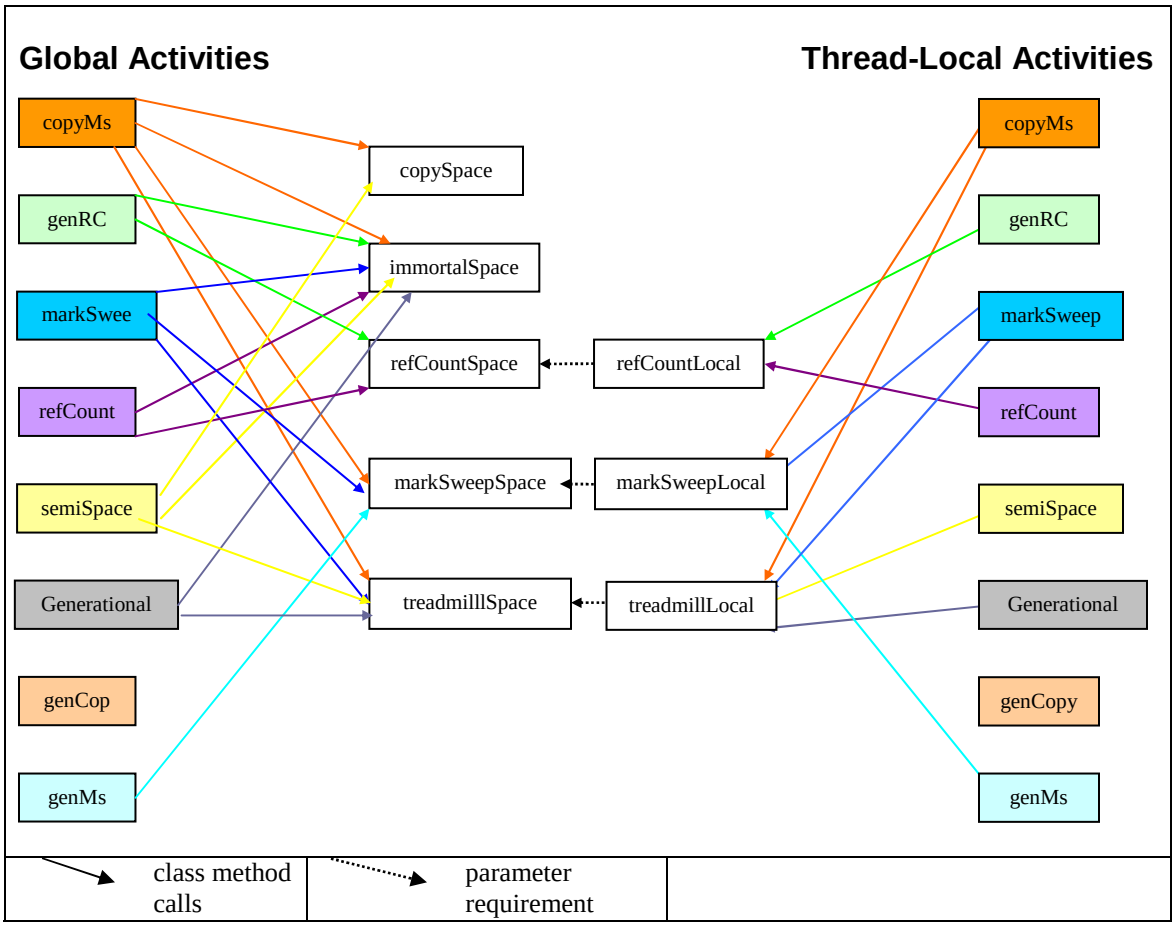

Figure 3. Control Flow of Plan through Policy

Given the structural properties outlined in Figure 3, Figure 4 further illustrates the symmetry of the crosscutting structure between the *prepare* and *release* methods in the global as well as local activity. That is, each policy that is used in prepare is also used in release. The only exception to this rule is in the `copyMS` plan's use of `copySpace` policy. The prepare method of `copySpace` is called in the global activity, yet the release of copySpace is not[1].

[1] This inconsistency may or may not be an error and will be considered in future work.

		Global					Local		
Plan		copy Space	immortal Space	refCount Space	markSweep Space	treadmill Space	refCount Local	markSweep Local	treadmill Local
Copy MS	p r								
Gen RC	p r								
MS	p r								
Ref Count	p r								
Semi Space	p r								
Generational	p r								
Gen Copy	p r								
Gen MS	p r								

p – prepare method call | r – release method call | Note: genCopy and genMS inherit policy from Generational

Figure 4. Policy Crosscutting Plan

Figure 4 also reveals symmetry between policy related global and local activities of each plan. In the figure, the related policy classes are assigned the same colours, with the local being a shade lighter than its related space parameter. This shows that each plan uses related policy classes in their global and local activities. This exposed relationship illustrates the parallelism between the control flows in the global and thread-local prepare methods as well as between the control flows in the global and thread-local release methods.

Refactoring this code to re-introduce policy to plan using aspects better exposes both the symmetry between prepare/release and global/local phases of the protocol, and better captures the internal structure of policy itself. Figure 5 shows the general structure of the design of these aspects. Developers can easily assess similarities and differences between plans with respect to memory management policy. Some plans such as *genRC* and *refCount* in the right half of Figure 5, can share the same aspect. Figure 6 shows the actual static aspect in our prototype to date.

When plan is *copyMS*

and on **globalPrepare** path
copySpace prepare
immortalSpace prepare
markSweepSpace prepare
treadmillSpace prepare

and on **threadLocalPrepare** path
markSweepLocal prepare(*markSweepSpace)*
treadmillLocal prepare(*treadmillSpace)*

and on **threadLocalRelease** path
markSweepLocal release(*markSweepSpace)*
treadmillLocal release(*treadmillSpace*)

and on **globalRelease** path
immortalSpace release
markSweepSpace release
treadmillSpace release

When plan is *genRC* or *refCount*

and on **globalPrepare** path
immortalSpace prepare
refCount prepare

and on **threadLocalPrepare** path
refCountLocal prepare(*refCountSpace*)

and on **threadLocalRelease** path
refCountLocal prepare(*refCountSpace*)

and on **globalRelease** path
immortalSpace prepare
refCount prepare

Figure 5. Structure of Dynamic Aspects

Currently the Jikes RVM allows users to specify the type of garbage collection they would like at the time of configuration. This option supplies a great testbed, but the overhead of reconfiguring and recompiling with each change of collector plans is tedious and time consuming. In the original implementation, fine-tuning of policy is not configurable at all. The addition of dynamic aspects into the system code still allows the convenience of specifying a chosen garbage collection plan, but the required rebuilding after each change in garbage collector plans still exists. By loading in the aspect involved in garbage collection at run-time the convenient garbage collection selection is allowed, without the need for repetitive rebuilding of the whole system. This arrangement could also allow on-the-fly garbage collection plan selection. There also lies the possibility of a system that determines the most compatible garbage collection plan and dynamically switches to that plan. This concept of loading aspects at run-time also has its disadvantages some of which include increased complexity and increased work for the system.

```
package com.ibm.JikesRVM.memoryManagers.JMTk;

privileged aspect PolicyAspect {

    private int state = 0;
    private final int GLOBAL_PREPARE = 0;
    private final int LOCAL_PREPARE = 1;
    private final int LOCAL_RELEASE = 2;
    private final int GLOBAL_RELEASE = 3;

    after(Plan p):target(p) && (execution(* Plan.globalPrepare(..))
                  || execution(* Plan.threadLocalPrepare(..))
                  || execution(* Plan.threadLocalRelease(..))
                  || execution(* Plan.globalRelease(..))) {

        switch(state){
            case(GLOBAL_PREPARE):
                CopySpace.prepare();
                Plan.msSpace.prepare();
                ImmortalSpace.prepare();
                Plan.losSpace.prepare();
                state++;
                break;

            case(LOCAL_PREPARE):
                p.ms.prepare();
                p.los.prepare();
                state++;
                break;

            case(LOCAL_RELEASE):
                p.ms.release();
                p.los.release();
                state++;
                break;

            case(GLOBAL_RELEASE):
                Plan.losSpace.release();
                Plan.msSpace.release();
                ImmortalSpace.release();
                state = 0;
                break;
        }
    }
}
```

Figure 6. PolicyAspect.java

5 Conclusions and Future Work

Dynamic aspects would allow developers to couple structural relationships with run-time configuration options, and we believe this combination would be particularly powerful in the context of garbage collection research. Our reasons are twofold. First, additional support for crosscutting concerns makes the internal structure of policy more clear, and makes the interaction between plan and policy explicit. Second, the ability to dynamically configure garbage collection allows for more responsive experimentation, and enables more effective attempts at feedback-based, system-wide optimization.

References

[1] Brian Goetz. How does garbage collection work? Developerworks, October 2003. www-106.ibm.com/developerworks/java/library/j-jtp10283/

[2] Jikes Research Virtual Machine, IBM. www-124.ibm.com/developerworks/oss/jikesrvm/

[3] Jikes Research Virtual Machine User's Guide, IBM. www-124.ibm.com/developerworks/oss/jikesrvm/userguide/HTML/userguide.html

A Family of Aspect Dynamic Weavers

Wasif Gilani
Olaf Spinczyk
University of Erlangen-Nuremberg
Martensstr. 1
D-91058 Erlangen
Germany
{wasif, spinczyk}@informatik.uni-erlangen.de

Abstract

Aspect-oriented programming is today mainly promoting the approach of applying aspects statically by means of preprocessors. We do find some work towards applying aspects dynamically but only limited to specific environments and no work so far has been directed toward our domain of interest which is deeply embedded systems characterized by very small memory and power. Also the work undertaken so far does not take into consideration the family-based approach since we have drastically different environments according to resources and one should be able to build up his own dynamic weaver from the family of weavers best suited for his environment. The paper presents an approach to achieve dynamism in applying aspects and also towards developing family-based aspect weavers to this problem which would be best suited for any sort of environment rather than having specific environment restrictions.

1 Introduction

To get maximum reuse of code and design, it is necessary to achieve complete separation of cross cutting concerns from the main functionality of an application. Following the aspect-oriented approach, the main functionality of any application is normally captured in classes and the cross cutting concerns are captured in separate aspects. Thus, in general case, we can say that the software systems developed using aspect-oriented programming techniques consist of classes and aspects whereas classes mainly implement the main functionality of an application, for example, put and get functionality in the case of bounded buffer problem, managing stocks or calculating insurance rates. Aspects, on the other hand, capture cross cutting concerns like synchronization, tracing, persistence, failure handling, or communication. Aspects are basically used to implement global policies in a system.

In the beginning, most of the work was based on static weaving which means adding aspects specific statements at join points to the classes statically at compile time. Static weaving produces well-formed and highly optimized woven code whose execution speed is comparable to the code written without AOP. There are certain environments where it is needed to be able to change the global policies implemented through aspects during run-time. Thus static weaving is not sufficient for such applications because in static weaving approach it is difficult to later identify aspect-specific statements in the woven code. Thus, it will be time consuming to adapt or replace the aspects dynamically during runtime or sometimes not possible at all. Although this flexibility is not a requirement in all scenarios, there are scenarios where applications could benefit from it.

Dynamic weaving means that aspects can be applied or removed at any time during runtime. Thus dynamic weaving allows the integration between components and the aspects at run time, resulting in a system which is more adaptable and extensible. In short, in dynamic weaving

aspects can be added to the system on the fly and, thus, can help avoid re-compiling, re-deployment and re-start of an application [5]. There are cases where it is more useful and flexible to have dynamic weaving as compared to the static weaving. One scenario would be where a load balancing aspect [4] could replace the load distribution strategy woven before with a better one depending on the current load of managed servers. Therefore in certain cases survival of aspects at run time is necessary to allow them to adapt to suitable policies according to execution time information. Another example could be addition of debugging aspect to get some particular data from some long running embedded application. It would be more flexible and beneficial to just add this debugging aspect to the application while running and get the required information than to stop the application and restart it again. Another case could be of the tracing aspect which could be deployed dynamically in some software system where some malfunctioning has happened. Another case where dynamic weaving provides edge over static weaving is hot fixes in web applications [10]. A hot fix is an extension applied to a running application server to modify the behaviour of a large number of running components. Another interesting example is the adaptation of mobile devices [10] since mobile devices have very small memory and so the device is not able to have all the software components needed in various locations from the start and so it should dynamically acquire the needed functionality it needs in a certain location and discard when location changes. This functionality has often a cross-cutting character and so is realized as aspects.

This paper starts with analyzing different existing dynamic weaver infrastructures to systemically describe the variabilities of this domain. The section 3 of this paper discusses the idea of program family concept and applying this idea towards family-based development of dynamic aspect weavers. In section 4 design methodology of our approach will be discussed. Section 5 describes one specific case of constructing dynamic weaver from the selection of certain features from feature model. Section 6 concludes the paper.

2 Dynamic Weaver Infrastructure

A run-time system is needed to support the weaving of aspects dynamically. There is not much research work going on in the direction toward building dynamic aspect weavers in the C++ domain. We do find some work but mainly in Java which is not suitable for our domain of interest which is embedded systems characterized by very small memories. Also focus has been on the development of single application specific weavers rather than families of weavers so there is no chance to make an optimized use of these weavers for any particular environment. We propose to build a family of dynamic aspect weavers to cover a whole range of environments.
Existing dynamic aspect infrastructures can be differentiated according to the way the functionality class is bound to the dynamic aspect weaver and the varying support provided by these for dynamic weaving. So far there have been three main approaches namely proxy-based, interpreter based and binary code manipulation.

In the proxy based approach [6], a proxy is used to intercept the incoming requests to the functionality class. The role of so called weaving is performed by an object called AspectModerator. All the aspects are registered with this AspectModerator object and proxy uses AspectModerator object to evaluate the aspects for every method of the functionality class. When the request arrives in the system, it is intercepted by the proxy and if the request is for the creation of an aspect then the proxy first checks whether this aspect is already registered with the AspectModerator object. If not, the proxy calls AspectFactory to create an aspect. The proxy then registers newly created aspects with the AspectModerator object. In case of a request for method invocation, the proxy calls the AspectModerator object to evaluate the aspects associated with this invocation. In this framework approach focus is only on reuse. This is not applicable for embedded systems because of excessive use of the design patterns with expensive abstract classes and virtual functions.

In interpreter extension approach, standard interpreter is transformed to build a new interpreter (plug-ins). This approach is normally applied by using Java virtual machine (JVM) since in C++ domain no interpreter is used. The interpreter is extended to check for the events happening in the execution and on happening of the events; advice code is applied as methods. Prose [5] is based on locating support for weaving and unweaving of aspects directly in the JVM. It is a JVM extension which can intercept calls at run-time. It makes use of the Java virtual machine debugging interface (JVMDI). It provides a new API within the JVM for weaving aspects at run-time called Java virtual machine aspect interface (JVMAI) which is designed as a plug-in to JVM. Thus applications must run with Prose specific JVM. Axon [12] is also based on interpreter extension approach and makes use of the debugging interface of JVM. It is an ECA (Event-Condition-Action) rules inspired model. Pointcuts are described in terms of events (join points) and conditions, while advice is described as an action associated with events and conditions. Axon is also implemented as a plug-in to Sun's JVM. Events are generated by the JVM and the conditions are checked by Axon via run-time inspection with the JVM debugging interface and the actions are advices which are implemented in plain Java methods. In this model around advice (AspectJ [2], AspectC++ [1]) is not feasible because it is based on interception at join points and there is no way an advice could replace the called method. Axon supports limited join points namely method entry/exit, exception throw/catch and field read/write. Both Prose and Axon are slow as the debugger imposes a certain overhead in the execution of applications. The JVMAI approach in Prose does not support crosscuts that add new members to a given class in the original code (introductions), because its implementation cannot change the source-code or byte-code of the original application. The debugger in current implementation of Prose is no longer available for use for debugging applications. Some of the approaches based on JVMDI are being modified to make use of just-in-time compiler instead of debugging architecture to improve the performance. Prose2 [10] is implemented by extending the VM's just-in-time compiler and no longer uses debugger architecture.

The binary code manipulation approach has mainly been employed in Java (JAC [7], Wool [11]). Most of these available approaches make use of JVMDI or changing the byte code at load time. Hooks are either inserted statically in all join points or inserted into the program at run-time "just-in-time" when the programmer directs the program to start using an aspect. Approaches where JVMDI is employed, the debugger interface allows a user to stop and query the state of a running program. These techniques are slow as debugger imposes certain overhead (context switches) on the execution of applications. Moreover these different approaches which make use of the JVM vary in terms of the join point support and performance. Wool [11] supports only method calls, field accesses, object instantiation, and exception handlers and does not allow introductions. JAC changes the byte code of classes of an application when they are loaded into the JVM and so have a high number of empty hooks and only support method calls and exception throws. In the C++ programming domain we are not aware of any previous work in aspect languages which provides dynamic weaving except for the microDyner [3] approach which was developed originally for the C language. This approach is processor specific (Pentium architecture) and does not allow more than one aspect to affect the same join point. In the microDyner approach, which makes use of binary code manipulation, aspects are deployed dynamically at run time in C programs. It realizes the weaving process by directly rewriting the code being executed. The microDyner approach is being extended to support C++ [8]. However, still this approach supports one aspect per join point and also aspects are not objects in this approach and so we cannot make use of nice features of object-oriented programming like constructors, inheritance etc.

These different dynamic weaving approaches differ from each other in a number of ways. Some of these approaches offer a limited set of join-points, thereby limiting the amount of application features an aspect can adapt. Some support just code join points while others support

introductions as well. Also these different approaches differ in terms of support for either single or multiple aspects per join point and advance knowledge of aspects etc.

In this section different existing dynamic weaving approaches have been analyzed to find out their commonalities and variabilities. All of these different approaches concentrate on satisfying the requirements for particular systems and offer different performance penalties like execution speed, memory consumption and join point support etc. For example, proxy-based weaver is not suitable for domains where there is a very small amount of memory and run-time resources because of extensive use of abstract classes and virtual functions. Also for such domains like embedded systems, which are very short of memory and run-time, it is not possible to use JVM based weavers because of the memory space occupied by the JVM. Program family concept implements the idea of building application specific systems and uses feature domain analysis to represent the commonalities and differences between the applications in a whole domain. In the next section, program family concept will be described along with its application in the field of aspect dynamic weavers.

3 Applying Program Family Concept

The main aim of our work has been to shift focus from the development of single dynamic weavers to the families of weavers. A set of programs is considered to be a program family if they have so much in common that it pays to study their common aspects before looking at the aspects that differentiate them [14]. Domain engineering [9] helps us to accomplish this goal. Domain engineering moves the focus from the code reuse to reuse of the analysis and design models. Domain analysis is the first phase of domain engineering which involves the process of systematic organization of the existing domain knowledge in a way that enables and encourages the extensions to be made in creative ways. The next phase of domain engineering is domain design which involves the development of an architecture for the family of systems in the domain and to devise a production plan. The last phase of domain engineering is domain implementation which involves implementing the architecture, the components, and the production plan using appropriate technologies. The results of the analysis, collectively referred to as a domain model, are captured for reuse in future development of similar systems. A domain model is an explicit representation of the common and the variable properties of the systems in a domain. Feature models define a set of reusable and configurable requirements for specifying the systems in a domain. A feature model consists of a feature diagram and some additional information. Features and feature models [9] are used to capture the commonalities and variabilities of systems in a domain during domain analysis. A key part of feature model is a feature diagram. A feature diagram represents a hierarchal decomposition of features including the indication of whether or not a feature is mandatory (each system in a domain must have certain features), alternative (a system can possess only one feature at a time) or optional (a system may or may not have certain features).

Some applications may require only a subset of services or features that other applications need. These ‘less demanding’ applications should not be forced to pay for the resources consumed by unneeded features [14]. In a program family concept a minimal subset of system functions provides a common platform of fundamental abstractions. These minimal subset of functions capture common functions that are useful to build specialized systems. The mechanisms from which more enhanced system functions can be derived are called minimal basis. A stepwise functional enrichment of the minimal basis is performed by means of the minimal system extensions. These extensions are made on the basis of an incremental system design, with each new level being a new minimal basis for additional higher level system extensions. Since the extensions are made only on demand thus a true application oriented system evolves.

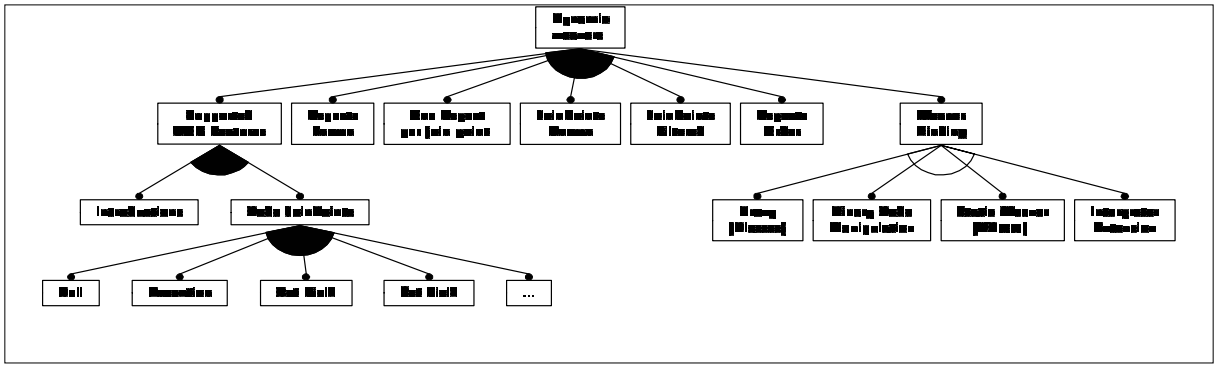

Figure 1. Feature model for Dynamic Aspect Weavers

Different applications can have different requirements from the dynamic weavers. We hold the view that less demanding applications should not be forced to pay for the resources consumed by the unneeded features. It is hard to imagine if any of the existing dynamic weavers could be used in embedded systems where applications are executed under extremely limited resource constraints. Thus weavers are required to be designed to specifically support the execution of applications under such resource constraint environments. In consideration of the specific demands of these applications, it becomes extremely difficult, if not impossible, to successfully adapt existing weavers. The program-family concept is being applied to create application specific dynamic weavers. A family-based dynamic weaver targets a wide range of applications including embedded systems. The program family concept does not dictate any particular implementation technique. In our proposal, based on program family concept, the dynamic weaver is incrementally enriched by minimal extensions. The weaver extensions are customized with respect to specific user demand. Thus applications are not forced to pay for the resources that will never be used. One of the main reasons for applying family-based design is to achieve the desired application orientation and to reduce the memory and run time consumption. The goal is to support applications with their desired specialized family member which provides all necessary functionalities but omits any features which are not required.

In the rest of this section domain analysis is performed for dynamic aspect weavers and a feature model is drawn to capture the commonalities and variabilities of dynamic weavers.

3.1 Dynamic Weaver Construction Set

We have made an effort to develop a feature model for dynamic aspect weavers consisting of a number of features. The output from this feature model is a family member which would be a dynamic aspect weaver. Use of feature model makes it easy to generate the family members and to handle the complexity of configuration in a better way. A feature model for “Dynamic aspect weavers” is shown in Figure 1. It is a quite simple model and allows specifying the required binding mode of the aspect weaver from the four available binding modes. Whether a dynamic weaver’s binding with functionality class is proxy-based, binary code manipulation, interpreter extension or static weaver based is up to the specific application requirements. In any dynamic aspect weaver it is assumed that only one of these binding possibilities would be used as shown in the feature model. The alternative features for dynamic weaver are indicated in figure by an arc which is not filled. Then after there can be lot of different scenarios regarding requirements for specific applications about before hand knowledge of aspects, aspects not known in advance, join points, order of activation of aspects, join point filtration, supported join points etc. Keeping in view of different possible requirements different dynamic weavers can be constructed by the

selection of different features. In the next sections all these different possibilities will be discussed along with the possible feature selection scenarios.

3.1.1 Aspects Known

There can be variability in the dynamic weaver construction regarding advanced information of aspects going to affect the join points. This feature "Aspects Known" from the feature model is selected in case when, in certain applications, it is possible to have an advanced knowledge of the aspects. In general case when this feature is not selected then there will be checks for all join points to find out if there are aspects registered for them irrespective of the possibility that for some join points there might not be any aspects registered at all. So in case even no aspect is registered, all checks, whether or not the system should execute advice, are performed. This situation will result in unnecessary method calls which involve large overhead.

It is possible to get rid of these types of checks for join points for which there are no aspects registered by selecting the feature "Aspects Known" in the dynamic weaver construction. This feature selection means that there is advance knowledge of aspects. When aspects are known in advance it would be much more efficient to only register the join points which are going to be affected by these aspects than to register all the join points and in result resources can be saved. Join points which are not going to be affected can be inlined. Now in this dynamic weaver construction the runtime infrastructure will be consumed for limited join points which are registered with the run-time system (only for which there are aspects registered) and so the system will be much more efficient.

If there are more aspects which are going to affect the same join point then normally lists are maintained against each join point containing ***before*** and ***after*** advices. So when there is an advance knowledge about the number of aspects going to affect each join point then it would also be possible to fix the size of advice lists associated with each join point and hence saving space. Another important benefit from the advance knowledge of aspects is that their order of execution can be resolved statically and so run-time infrastructure can be saved. Example where this can be implemented is having some system and there are three policies of security to be implemented by means of aspects. Also if we have an application in which more than one aspect can affect the same join point then it means it is essential to select the "Aspect Order" feature as well, while dynamic weaver construction, to resolve the conflicts between different aspects affecting the same join points.

3.1.2 Aspects Order (Interaction)

There are some dynamic weavers which are restricted to support only one aspect per join point [8]. In other cases we have frameworks [6] which support any number of aspects affecting the same join point. If more than one aspect (advice) affects the same join point and there is dependency between the advice codes then it might be necessary to define an order of advice execution ("aspect interaction") in order to avoid conflicts. The order of activation is supported in static weaving technologies like AspectC++, AspectJ. In Netinant's aspect-oriented framework, the order of activation of aspects is predefined, it is defined that the synchronization aspect has to be verified before the scheduling aspect. If security aspect is introduced then it is needed to be handled before the synchronization aspect. A possible reverse in the order of activation of the aspects can violate the semantics. Moreover it is possible to alter the order of activation on the fly.

In certain dynamic weavers, like microDyner, it is not allowed that more than one aspect can affect the same join point. Thus in such dynamic weaver constructions there is no need to select the feature "Aspects Order" which is only required when dynamic weaver is supposed to support multiple aspects per join point. Also in such cases there is no need to maintain lists of before and after advices against each join point registered with the run-time system and so there would be no

need to traverse the whole lists during runtime to invoke advices dynamically and as a result framework will be much more efficient and fast.

3.1.3 Join Points Known

In the case of join points there can be again two possibilities, first being that all affected join points are known in advance and second that there is no advance knowledge of the join points going to be affected by the aspects. In the case of join points known in advance, once the system starts running it will be affected by the aspects which are already into the system. If the system is extended (classes are loaded incrementally) then the aspects are not able to affect these additional loaded classes (aspects do not have to affect the code loaded later in the system). In other words, additional classes are allowed to be loaded if it is known that no aspect would be affecting join points in these classes. When join points to be affected are known in advance, compile time matching can be done resulting in saving of run time resources. In example we can think of a system in which we know join points which are going to be affected in advance and so we are able to apply different versions of security policies, implemented as aspects, to this system. Now if the system is extended, aspects would not be affecting the additionally loaded classes.

3.1.4 Join Points Filtered

Join points can be filtered, for example, by means of pointcuts according to the varying requirements. In some systems there might be requirements to apply aspects in specific modules. For example we can have two systems, in one system aspects can affect the whole system but in the other case we can do filtration and so only specific module will be affected by the aspects. There is no need to change the aspects, only the behaviour of the aspect is modified but the implementation remains same.

3.1.5 Supported AOP Features

To which join point in a system is it possible to apply aspects? There are different approaches and these vary either according to the type of join points supported by them or if they support introductions. Code join points can be defined as method calls, method executions, set field, get field etc. *Introduction* is how modification is done to a program's static structure, namely, the members of its classes and the relationship between classes. Introductions are used to extend program code and data structures in particular. Most of the join points are supported by AspectJ and AspectC++ like method call, method execution, constructor call, constructor execution, object initialization etc. Both AspectJ and AspectC++ also support introductions. There are other approaches which support specified set of join points like Wool [13] supports only method calls, field accesses, object instantiation, and exception handlers. Axon [12] supports method entry/exit, exception throw/catch and field read/write join points. JAC' supports method calls and exception throws [7]. Wool does not allow introduction since the HotSwap does not allow reloading a class file to which a new method or field is appended. Similarly Prose [5] does not support introductions. There are certain applications where it might be required to have support for introductions and thus dynamic weavers would need this feature selection. One example where introduction could be needed to have support is when pointers could be added to objects statically and then during run-time data could be added to these objects.

This is still not a very comprehensive feature model. We are still working on it and there are many more common and variable features of dynamic weavers which are needed to be represented in this feature model.

4 Design Methodology

Our proposal is based on promoting two ideas. First a dynamic weaver should be able to make use of both the static and dynamic weaving according to the specific application requirements and cost considerations. Secondly the design should be based on the family-based approach. The reason for having support of static weaving in the dynamic weaver construction is logical if we consider the advantages we get from static weaving in terms of performance of static weaving as compared to dynamic weaving. Ideally, an implementation should support both. Aspects that don't need to be adapted at runtime should be woven statically for performance reasons since our dynamic aspects do consume run time resources and so dynamism should be allowed for aspects which do have runtime changing behaviour or which need to change policies during runtime. Even in the case of byte code manipulation, users can be allowed to choose a suitable hook at each join point considering the whole cost. Either they can be inserted as a breakpoint by a debugger and so executed by debugger or can be embedded as a method call using dynamic code translation [12]. For example an application where we could think of having support for both static and dynamic weaving together is when an aspect such as authentication [13] can be woven statically as it is very unlikely that an authentication policy changes during program execution. On the other hand, a scheduling policy has to be adapted most likely at run-time. Scheduling can therefore be viewed as a dynamic aspect. A good mix of the dynamic weaving and static weaving promises to improve AOP effectively. Aspects can be efficiently tested by dynamically inserting them, checking the behavior of the application and then removing the aspects to perform corrections. In case of adding an aspect statically it is required that the running application be shut down thereby losing all the run time data. Eventually once the aspect code is stable, the aspect can be woven through the application code using a static weaver to improve performance [5].

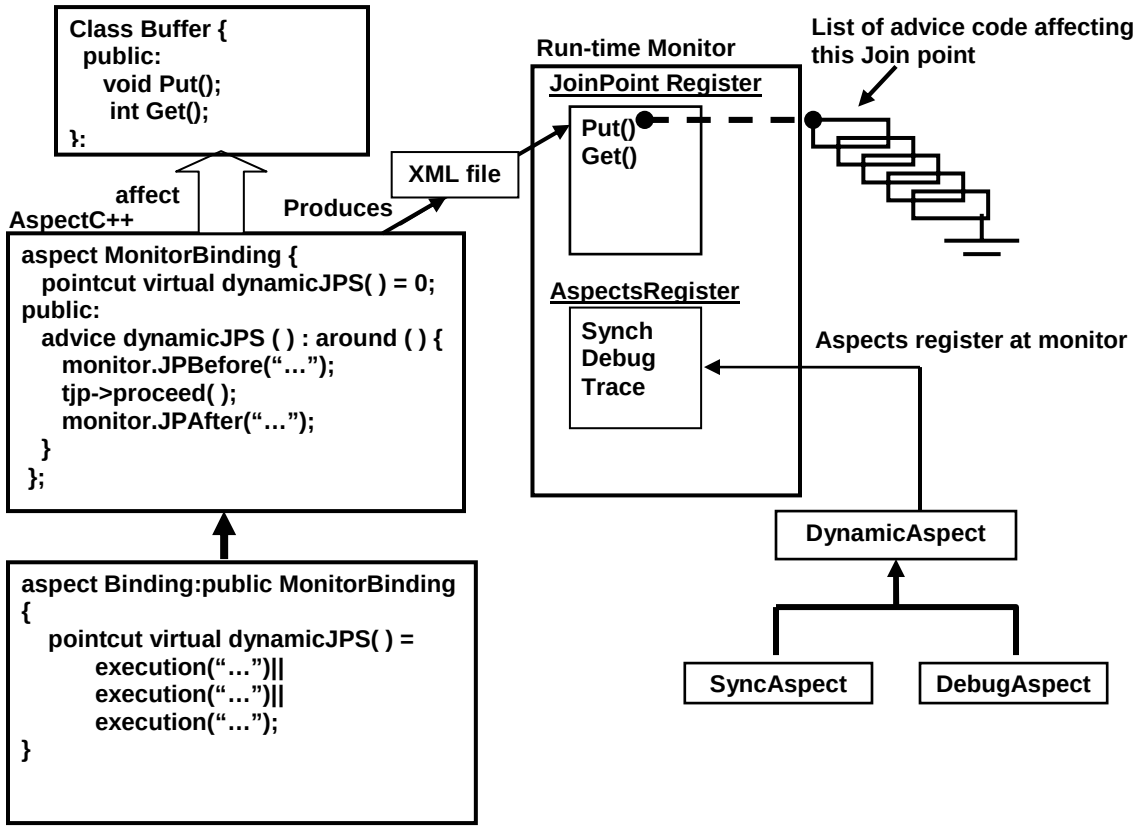

Figure 2. Aspect Dynamic Weaver Architecture

5 Weaver Instantiation from Family of Dynamic Weavers

In this section an instantiation (construction) of a dynamic weaver will be described from the family of dynamic weavers (feature model). This weaver would be constructed in way that it would be able to support static as well as dynamic aspects and also it would be able to support multiple aspects per join point. As can be seen from the feature model, there is a limit to select only one binding mode feature of the dynamic weaver to the functionality class. This particular construction makes use of selection of feature "static weaver (AspectC++)" as a binding mode to the dynamic weaver (Figure 2). Though AspectJ is also an option which is a complete and powerful language extension for aspect-oriented programming but the costs (run- time and code size) of Java run-time-environment are not feasible for deeply embedded systems which are having very low memory constraints. The selection of binding feature is totally independent of the functionality class. Dynamic aspect weavers can be constructed by selecting any of the binding modes available. In our construction a dynamic weaver is constructed from three main modules which are independent of each other and have clearly defined interfaces. These modules are:

- Run-time monitor
- Aspect binding (Dynamic Aspects)
- Weaver binding (Static Weaver)

The run-time monitor plays a central role in this dynamic weaver. The main role of the run-time monitor is to register the join points and the aspects. It also co-ordinates interaction between the aspects and the functionality class. In this dynamic weaver construction, it is assumed that more than one aspect is able to affect the same join point and so the feature "Aspects Order" is selected. The source file for run-time monitor for registering join points is an XML file which is generated by the static weaver (AspectC++). This file could be used statically to register join points in the case join points are known in advance. In embedded systems one rule is followed which is to do as much processing as possible before run time, creating a run-time environment that is as efficient as possible. In case of join points not known in advance we can convert this XML file to a binary format to make the processing efficient and thus improving the system's performance. The run-time monitor also has a task to take care of order of execution of aspects in the case if there are more than one aspect interested in the same join point.

There are two types of aspects which are supposed to be supported by this dynamic weaver. First being the static aspects defined in a specialized aspect description language like AspectC++ or AspectJ. Secondly, the aspects which are dynamically invoked during run time and are C++ classes. Each class is supposed to have two advices which are simple methods. One is "**JPBefore(…/*method id */…)**" and other is "**JPAfter(…(…/*method id */…)**" and both of these advice methods take method identity as a parameter and it is the run-time monitor which invokes these methods in static weaver.

The static weaver (AspectC++) has been developed by the authors and it is a general purpose aspect-oriented extension of C++, modeled following the approach of AspectJ. AspectC++ is implemented as a C++ preprocessor based on PUMA [1]. PUMA is a source code transformation system for C++. The output of this preprocessor is the C++ source code with the aspect code woven in. Afterwards a conventional C++ compiler is used to get the code translated to the executable code.

In this aspect dynamic weaver, the functionality class is bound to the framework with a static weaver (AspectC++). The static weaver produces XML file as a result of the interaction with the functionality class. This XML file consists of all the information of the join points contained in the functionality class. These join points are then registered with the run-time monitor using this generated XML file. This dynamic weaver is being constructed on the idea that one should be

able to decide which aspects need to be woven dynamically at run-time and which aspects to be woven statically at compile time depending on the performance. Thus there are certain aspects which don't need to be woven dynamically and so static weaver is used to weave these aspects statically to save run-time infrastructure. The dynamic aspects are normally simple classes and they are registered with the run-time monitor. As soon as some dynamic aspect is registered with the run-time monitor, the list of join points registered with the run-time monitor is traversed to find out which join points are affected by this aspect. Since in this dynamic weaver one join point is supposed to be affected by more than one aspect (here again other scenarios are possible by the selection of, for example, "One Aspect per Join point" feature), a list of advices, which are methods defined in each aspect, which are to affect a certain join point, is maintained to be executed once this join point is invoked by the run-time monitor in the static weaver.

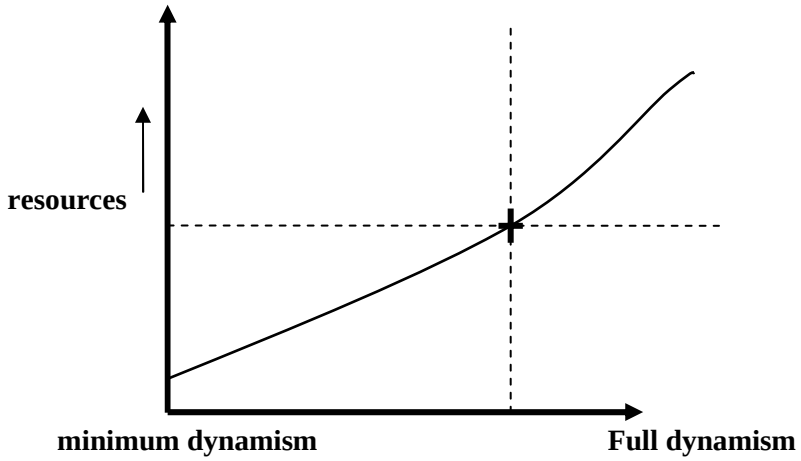

Figure 3. Resource consumption vs. provided dynamism

In this dynamic weaver, run-time monitor is able to add or remove aspects on the fly. Functional classes do not know about the aspects in advance but of course it purely depends on the features selected from the feature model (Figure 3). When some major dynamic aspects of the system are to be defined like scheduling, synchronization, tracing, fault tolerance, these aspects are defined as being derived from the same super-aspect. Thus this dynamic weaver promotes reusability. An abstract super-aspect provides transparency since sub-aspects can use the super-aspect without knowing the internal implementation details of super-aspect. Even new aspects can be introduced into the system without any problem. If some super-aspect is changed to add some new features, the sub-aspects don't need to be changed as far as the interface remains constant. All dynamic aspects are registered with run-time monitor with some call like:

```
monitor.registerAspect(Aspect SyncAspect)
                       /*SyncAspect captures synchronization
concerns */

monitor.JPBefore(putId);  /* all before  advices are executed
                             when run-time monitor is invoked
                             from static weaver */

tjp->proceed();           /*actual method is invoked */

monitor.JPAfter(putId);   /* all after advices are executed when
                             run-time monitor is invoked from
                              static weaver */
```

In static weaver (AspectC++) there is also a possibility to derive an aspect from super aspect. Abstract aspects can be defined from which new aspects can then be redefined through inheritance, thus providing programmers with an aspect hierarchy. Using advice "around" feature of static weaver allows to first invoke before advice codes `(monitor.JPBefore(…/*method id*/…))` associated with any join point, then invoking the join point itself (`tjp->proceed`) and then finally invoking the after advices (`monitor.JPAfter(…(…/*method id*/…)`) of the join point.

6 Conclusion

This work provides a base for developing application specific dynamic weavers. Instead of inventing a new dynamic weaver architecture, this approach provides the user with the ability to construct many of those architectures. The concept of program family has been applied to build a family of aspect dynamic weavers. A feature model has been built which provides an abstract, concise and explicit representation of the commonality and variability present in the domain of dynamic weavers. Using this approach it is possible to build dynamic weavers with as much functionality as one application can afford. This work promotes the idea that one should not be asked to suffer for the services he does not require. The program family concept helps to create featherweight weaver abstractions. These abstractions can be used by the user to construct a number of dynamic weavers.

The main goal of this approach is to be able to construct application-specific dynamic weavers by selecting only those features from the feature model which are required. The example of a specific dynamic weaver instantiation from a family of dynamic weavers has clearly demonstrated that it is possible and feasible to construct a weaver according to the specific requirements of a particular application. Figure 3 illustrates that as we move towards more dynamism in building the dynamic weavers, we have to pay more in terms of resources. The goal, while constructing a dynamic weaver for any application, is to get to a point as shown in figure where we are not exhausted of the resources and we have as much features added to our weaver construction, to support dynamism, as possible. A simple example where we could think of constructing customized dynamic weaver would be of some embedded system with very small memory in the range of, for example, 30 Kbytes. Now while doing application specific construction of a dynamic weaver for such systems, where the join points are normally known in advance, we can select features from the feature model to have as much degree of dynamism as memory space allows. The result would be a dynamic weaver which would be able to fully utilise the available memory space and allow us with as much dynamism as we can afford.

References

[1] Spinczyk Olaf, Gal Andreas, Preikschat Wolfgang Schroeder, *AspectC++: Language Proposal and Prototype Implementation*. In OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa, USA, Oct. 2001.

[2] Xerox Corporation, AspectJ programming guide, Online documentation, http://www.aspectj.org, 2003.

[3] Marc Ségura-Devillechaise and Jean-Marc Menaud and Gilles Muller and Julia Lawall, *Cache Prefetching as an aspect: Towards a Dynamic-Weaving Based Solution*, in Proceedings of the 2nd international conference on Aspect-oriented software development, p. 110-119, ACM Press, Boston, Massachusetts, USA, Mar 2003 Web

[4] Frank Matthijs, Wouter Joosen, Bart Vanhaute, Bert Robben, and Pieere Verbaeten., *Aspects Should not Die*, Position paper at the ECOOP '97 workshop on Aspect-Oriented Programming.

[5] A. Popovici, T. Gross, and G. Alonso., *Dynamic Weaving for Aspect Oriented Programming*. In 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, Apr. 2002.

[6] C. A. Constantinides, T. Elrad, M. E. Fayad, Netinant P., *Designing an aspect-oriented framework in object-oriented environment*, ACM Computing surveys, March 2000.

[7] Pawlak, R., Seinturier, L., Duchien, L., Florin, G., *JAC: A flexible framework for AOP in Java*. In Reflection 2001

[8] Yan Chen, Masters thesis, *Aspect-Oriented Programming (AOP): Dynamic Weaving for C++*, August 2003, Vrije Universiteit Brussel and École des Mines de Nantes

[9] Krzysztof Czarnecki, Ulrich W. Eisenecker, *Generative Programming Methods, Tools, and Applications*, Chapter 4, Addison Wesley 2000

[10] A. Popovici, T. Gross, and G. Alonso., *Just-In-Time Aspects: Efficient Dynamic Weaving for Java.*, AOSD 2003, Proceedings of the 2nd international conference on Aspect-oriented software development, Boston, Massachusetts

[11] Yoshiki Sato, Shigeru Chiba, Michiaki Tatsubori, *A Selective, Just-In-Time Aspect Weaver*, Proceedings of the second international conference on Generative programming and component engineering, Erfurt, Germany, Year of Publication: 2003

[12] S. Aussmann, M. Haupt, *Axon – Dynamic AOP through Runtime Inspection and Monitoring*, First workshop on advancing the state-of-the-art in Run-time inspection (ASARTI), 2003.

[13] Shigeru Chiba, Yoshiki Sato, and Michiaki Tatsubori, *Using HotSwap for Implementing Dynamic AOP Systems*, ECOOP'03 Workshop on Advancing the State of the Art in Runtime Inspection (ASARTI), July 21st, 2003.

[14] D.L. Parnas., *Designing Software for Ease of Extension and Contraction*, IEEE Transactions on Software Engineering, SE-592):128-138, 1979.

Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System

Philip Greenwood
Lynne Blair
Computing Department, Lancaster University, Lancaster, UK
p.greenwood@lancaster.ac.uk | lb@comp.lancs.ac.uk

Abstract

As computational complexity of systems continues to increase, the amount of maintenance required to keep them operational will also increase. Autonomic systems have the aim of reducing the amount of maintenance required by performing certain levels of maintenance themselves. This paper outlines the case of using dynamic AOP to implement such a system. The benefits and issues arising from using dynamic AOP will be looked at and discussed.

1 Introduction

Computational complexity is related to the amount of time required to carry out a certain operation [5]. As computing power increases more operations are able to be carried out in the same amount of time and so system complexity will increase. As systems become more complex increasing amounts of maintenance are required to be performed on them in order to keep them operational. This requires vast amounts of money and time for skilled IT workers to maintain these complex systems.

What is needed is a system which still performs all the required complex tasks of modern systems but also does not need the high levels of maintenance that these systems require. It is unrealistic to build a system that requires no maintenance at all; instead autonomic systems are being developed in which systems are able to perform certain levels of maintenance themselves.

Automating the maintenance tasks will, theoretically, improve the availability of the system as human error is the most common cause of systems failure. As a result reducing the amount of human contact is likely to reduce the number of failures. The availability will also be improved as the system does not need to be taken off-line for changes to be made.

As well as reducing the risk of failures and the time spent performing maintenance, autonomic systems will also reduce the operating costs, which is a vital factor in the competitive times we live in. The most obvious saving is the reduced man hours required.

Autonomic systems are generally implemented by using a monitoring component to detect when an adaptation is required. The monitor will then trigger an event to reconfigure the system to suit the current operating conditions. The type of adaptation will depend on the past and present conditions; it could be as simple as changing some parameter or it could require a more complex adaptation where a whole component needs to be swapped to alter the behaviour.

The main aim of Aspect-Oriented Programming (AOP) is to improve the separation of concerns by encapsulating crosscutting concerns. We propose that AOP and more specifically dynamic AOP can be used to encapsulate the adaptations that are required to implement an autonomic system. Combining this property of encapsulation with dynamic AOP allows the adaptations to be neatly contained and be applied at run-time. Additionally, the majority of the concerns that need adapting will also be crosscutting, for example security, synchronisation,

caching; AOP will allow these concerns to be cleanly encapsulated. This paper presents a discussion of the key issues relating to the use of AOP for implementing an autonomic system.

The remaining sections of this report are structured as follows. Section 2 describes in more detail the properties of autonomic systems. Section 3 then looks at the various dynamic AOP techniques available and assesses their suitability for this implementation. Section 4 describes a caching example which illustrates how dynamic AOP can be used to implement a basic autonomic system. Section 5 briefly analyses this example then lists some problems of using AOP. Section 6 will then describe other complementary techniques that could be used and examine various other implementations of autonomic systems which use AOP. Finally, section 7 concludes the report and summarises this paper.

2 Autonomic Systems

As mentioned previously, autonomic systems are systems that are able to perform maintenance on themselves. They achieve this primarily by monitoring various variables relating to their running environment. When they detect that something is not behaving as intended, or the environment has changed in such a way that results in the system not running optimally, then they will perform some re-configuration on themselves to correct this.

2.1 Overview

IBM is one of the leading organisations developing autonomic systems; they have outlined four key properties that a system needs to posses to be classed as autonomic [8]:

- Self-Configuring – systems adapt automatically to dynamically changing environments
- Self-Healing – systems discover, diagnose and react to disruptions
- Self-Optimising – systems monitor and tune resources automatically
- Self-Protecting – systems anticipate, detect, identify and protect themselves from attack

For a system to be fully autonomic it is required to possess all of the above features; however a system can be developed with a sub-set of these features to be partially autonomic.

Each of these features will in someway alter the way in which the system will behave. Where the system requires high availability these changes will have to be performed dynamically without taking the system off-line. Hicks et al [6] describe how this can be done using dynamic patches without the need to use redundant hardware (a traditional approach to allow adaptations to be made to systems while maintaining system availability). This solution however, does not go far enough for autonomic systems; it is still the programmer who must select *which* patch should be applied and *when* it should be applied.

There are various implementations [3] [4] [17] [19] of autonomic systems using AOP concepts currently available and they all encapsulate the adaptations differently; we shall examine these implementations in section 6. We propose to use dynamic AOP to encapsulate the required adaptations and dynamic weaving to be able to apply these adaptations at run-time without needing to take the system off-line and without requiring redundant hardware.

2.2 Desired Properties

Next we list a set of fundamental properties required to implement an autonomic system. These properties relate to the process of applying the adaptations to the base-code, encapsulating the adaptations and specifying the relationships/dependencies. In this list we will reason why dynamic AOP is suitable for this use and justify our decision in using it

.

Apply Adaptations Dynamically

Dynamic AOP is a natural choice for implementing an autonomic system due to the nature of it being able to apply code retrospectively to a running application. However, one limitation is that new code can only be woven at the points in the base-code which fit the join-point model of the selected dynamic AOP framework and is limited to the types of advice the language allows (before, after or around). For the majority of cases this model will be acceptable.

Easily Remove Adaptations

Most dynamic AOP frameworks allow the easy removal of previously woven aspects. By simply issuing a command, the advice attached to a particular join-point can be removed. Also, as the byte-code is not modified, the system can be easily stopped and restarted to roll back to the original state of the system. Other problems can arise when this behaviour is not desirable which will be discussed later.

Encapsulate Adaptations

One of the main reasons why AOP has been chosen for this purpose was the fact that the majority of adaptations that are used in an autonomic system tend to be cross-cutting concerns such as caching, security, persistence, etc. AOP has been shown to improve the encapsulation of these types of concerns and also some types of functional concerns.

Specify Relationships

Some AOP frameworks allow, to a certain degree, relationships to be specified between aspects, for exmaple both JAC and AspectWerkz allow this. Both of these frameworks allow the order which aspects should be applied to the base-code to be specified. However, for autonomic systems these solutions do not go far enough. The AOP frameworks solve some problems but autonomic systems need more complex relationships to be specified which will include some sort of conditional relationships.

Implement Fine Grained Changes

All AOP frameworks allow some degree of modification to the method level of a system which is much finer grained than the majority of current autonomic systems. Most current autonomic systems are component based and only allow whole components to be swapped or adapted. This can be inefficient and time consuming to implement when only a small change to a single class is needed.

Apply Adaptations to Various Points in a System

The points at which aspects can be woven with the base-code needs to follow the join-point model of the selected framework. Fortunately, the majority of frameworks have a wide reaching join-point model. These models allow code to be attached to a wide variety of points during a programs execution from exception throwing to field access operations.

3 Dynamic AOP

Static AOP languages have been used to implement systems for sometime now and techniques such as AspectJ [12] are now gaining much maturity. However, dynamic AOP languages have recently started to appear and be widely used.

An aspect created using a dynamic technique is woven at run-time and provides the extremely powerful tool of allowing the application to be modified, by changing the aspects currently woven and weaving new aspects, while the application is still running. However, the downside to this is the performance hit taken due to checks having to take place to determine

whether an aspect should be woven at the current point of execution. Further problems arise regarding safety, security and compatibility.

There are currently many dynamic AOP techniques being developed, the majority of which are fairly immature so they still require more development and testing. At this stage in our development process it is not necessary for the chosen dynamic AOP framework to be 100% stable, as it is expected that no AOP framework will exactly fit the properties required so customisations will have to be made to the AOP framework to suit our needs which this paper will outline.

This next section will look at three dynamic AOP frameworks: JAC, AspectWerkz and Prose, each of these techniques possess the desired properties listed in section 2.2 to a lesser or greater extent. As well as describing the implementation details of these techniques and relevant information regarding these desirable properties will also be mentioned.

3.1 JAC

JAC [14] [15] [18] is an AOP framework developed by Renaud Pawlak which is implemented in pure Java and requires no language extensions.

JAC relies on BCEL [2] which is used to alter the byte-code of a Java object at class load-time. BCEL is used to manipulate the byte-code to add references to aspect *wrapping* methods that are used to advise the base-code. The wrapping methods are linked to form a *wrapping chain;* each method calls the next in the chain and if applicable the method is executed. It is the job of the last wrapping method in the chain to call the method which is being wrapped.

One of the problems of AOP which JAC aims to solve is *inter-aspect composition* which is related to the specification of relationships property. JAC allows the user to define a *composition aspect* which can be used to define certain relationships between aspects; a useful property when implementing an autonomic system. The creators of JAC have identified the following issues:

- Checking for aspect compatibility with the application
- Checking for inter-aspect compatibility
- Checking for inter-aspect dependency
- Checking aspect redundancy
- Ordering/selecting the aspects at run-time

The composition aspect allows the programmer to define a set of rules which can prevent the above issues from occurring. However, to prevent these issues from occurring the rules must be defined to be ‘sound and complete’; this is often difficult to achieve. The issues that JAC and the composition aspect aims to solve are similar to those of autonomic systems; decisions have to be made in order to choose the most appropriate adaptations/aspects.

JAC easily allows new aspects to be woven and later removed to existing or new joinpoints at any point during the system’s execution. Also JAC allows the common types of advice to be implemented such as after, before and around advice, so the required fine grained changes can be implemented.

One negative feature of JAC is that field access operations are not part of its joinpoint model, this limits where the adaptations can be applied to the system.

3.2 AspectWerkz

Similar to JAC, AspectWerkz [1], created by Jonas Bonér, allows the creation of dynamic aspects using pure Java and again, does not require any language extensions to implement the aspects.

Like JAC, AspectWerkz uses run-time modification of the bytecode but uses a modified classloader to weave the aspects with the base-code instead. AspectWerkz hooks directly into the

bootstrap classloader and can then weave aspects to any classes loaded by the preceding classloaders. A wide range of joinpoints are supported and AspectWerkz uses the same semantics as AspectJ which makes AspectWerkz very simple and easy to use.

Similarly to JAC, AspectWerkz also allows new advice to be attached to existing pointcuts at run-time, allowing new behaviour to be introduced, again a useful property for an autonomic system.

However, AspectWerkz does not allow new joinpoints to be created at run-time; this is a limiting feature for autonomic systems and limits the dynamic property of the framework. There are two ways around this problem. The first is to reload the classes in a new classloader, but this could result in loss of state information and would require having control of how the classes are loaded. The second solution involves creating a very 'generous' joinpoint which would cover every possible pointcut so that advice could be added at any possible joinpoint. This could introduce a large overhead when only a subset of the possible joinpoints are used.

Just as JAC allows a composition aspect to be defined to perform checks before advice is executed, AspectWerkz allows a similar module to be defined called a JoinPointController. This gives similar functionality as JAC's composition aspect and again allows certain relationships between aspects to be specified.

The joinpoint model of AspectWerkz is slightly better than the one of JAC as AspectWerkz allows field access operations to be specified. This allows the adaptations to be applied to a wider range of point much more easily.

3.3 Prose

Prose [16] developed by Popovici et al is another dynamic AOP framework which again, like the previous two examples, does not require any language extension as the aspects are implemented in pure Java. The major difference between Prose and the previous two approaches is that Prose aspects are woven at run-time using JIT compiler weaving.

Prose relies on two main modules, the Execution Monitor and the AOP Engine. The execution monitor is responsible for activating new joinpoints and notifying the AOP engine when a joinpoint is reached. The purpose of the AOP engine is to decompose aspects into joinpoint entities and activate them with the execution monitor. The AOP engine is also responsible for executing advice when notified by the execution monitor that a joinpoint is reached.

The creators of Prose have identified four requirements of an AOP framework that they wish to meet:

- Efficiency under normal weaving
- Secure and atomic weaving
- Efficient advice execution
- Flexibility

One of the most relevant properties listed above when implementing an autonomic system is secure and atomic weaving. This is important for several reasons but this is a feature not supported in JAC or AspectWerkz. These issues will be discussed in more detail later. Prose does offer a further advantage over JAC and AspectWerkz through its improved performance due to run-time weaving.

As with AspectWerkz similar difficulties arise in Prose when new joinpoints need to be added, although advice can still be easily added or removed from the Prose framework. Again, this limits its dynamic properties and its effectiveness.

There are also two disadvantages when compared to other languages; it does not allow any specification of the relationships between aspects as seen in JAC and AspectWerkz. Additionally,

Prose does not allow ‘around’ advice to specified, unlike AspectWerkz and JAC, which is a problem when the behaviour of a method needs to be replaced. This limitation will hamper the development of an autonomic system as fine grained changes cannot be implemented as easily.

The joinpoint model which Prose uses is not as extensive as that of AspectWerkz, but it still allows advice to be attached to method entry/exit events and field access operations. Again, this allows the adaptations to be applied at a variety of places in the system.

As can be seen from these descriptions of a selection of dynamic AOP frameworks, each have their own advantages and disadvantages. Whichever framework is chosen, some modification will be needed to the selected framework to implement all our desired properties.

4 Automated Cache Example

This section will briefly describe a simple example illustrating how an autonomic cache concern has been implemented using JAC. JAC was chosen due to it being the most familiar technique to us at this time and the fact that it permits the addition of new joinpoints dynamically. However, the decision to use JAC for the final autonomic system has not yet been made; it is only being used at this stage to simplify the process and for evaluation purposes. This example will highlight some of the properties described above.

The example will show how a caching aspect is added autonomically and dynamically depending on the current environment conditions. The base-code will simulate a request-response operation where the server object sending the reply will introduce a delay by sleeping for an ever increasing time. Once this delay has reached a threshold, the caching aspect will be introduced to improve performance.

```
1 public String request(int req){
2   Thread.sleep(time);
3   switch (req){
4     case 1: return "One";
5     case 2: return "Two";
6     case 3: return "Three";
7     case 4: return "Four";
8     case 5: return "Five";
9     default: return "Not valid";
10  }
11 }
```

Figure 1. Server Code

The code segment above shows the server code. When the request method is called an integer is passed to the method and the string value of that integer is returned. Line 2 is responsible for introducing the delay; the variable time is gradually increased to simulate network delay so that the response takes longer to be received.

```
pointcut("ALL","ALL","request.*","MonitorWrapper","monitor",null,
false);
```

Figure 2. . Monitor pointcut

The above construct creates the pointcut to monitor the response times from the server. The first three parameters specify the point in execution where the advice should be specified. They specify that any method named ‘request’ that is located in any class with any number/type of parameters should be the pointcut. The following two parameters specify the method that should be used as the advice and the class its located in, in this case the class is MonitorWrapper and the method is monitor. The last two parameters are not relevant to this example. One problem which arises when defining such a joinpoint is that it will not be known at run-time where the aspects should be applied to. A potential solution is proposed in [10] which uses reflection and parameterisation of the joinpoints to generalise them.

```
1 public Object monitor(Interaction interaction throws Error{
2   long beforeTime= System.currentTimeMillis();
3   Object obj= proceed(interaction);
4   long afterTime= System.currentTimeMillis();
5   long duration= afterTime-beforeTime;
6   if (duration>threshold&&!weaved){
7    weaved=true;
8    Jac.remoteWeaveAspect("CacheTest","s0","CacheAC",
     "c:\\jac\\CacheTest\\cache.acc”);
9   }
10  return obj;
11 }
```

Figure 3. Monitor advice

The above monitoring code is a piece of around advice which surrounds the request method described earlier and is woven when the application is started. Lines 2, 4, and 5 are used to calculate the time taken to execute the wrapped method that is called in line 3. Line 6 implements an if-statement to determine whether the threshold has been reached, if it has then the caching aspect is woven in line 8. The appropriate method from the JAC package is called and parameters are passed to identify the application and the server which the aspects should be woven to. Additionally, the class which implements the aspect and the path to the aspect configuration file are also passed.

```
pointcut(".*","ALL","request.*","CacheWrapper", "checkcache",null,
false);
```

Figure 4. Caching pointcut

Again the above code segment declares a pointcut, which specifies where the new caching aspect should be woven to. The request method is again specified as the pointcut, but this time the method will be wrapped by the checkcache method in the CacheWrapper class (see below).

```
1 public Object checkcache(Interaction interaction) throws Error{
2  Integer arg= (Integer)interaction.args[0];
3  Object result= cache.get(arg);
4  if (result==null){
5   result= proceed(interaction);
6   cache.put(arg,result);
7  }
8  return result;
9 }
```

Figure 5. Cache advice

The main aim of the above code is to intercept the calls to the request method, extract the parameter passed to it and examine the cache to see if that value is already stored in the cache. Line 2 gets the parameter passed to the request method, the following line then tries to retrieve the cached value. An if-statement on line 4, checks whether the retrieved value is null or not. A null value here indicates that there is no cached result for the value passed and so the request method needs to be called anyway, this is done so on line 5. The value which is returned from the request method is now cached for later use. Line 8 then returns the correct value to the method caller; this will either be the value retrieved from the cache or the value returned from the request method.

As mentioned at the beginning of this section this is only a simple example; many important issues have been missed for the sake of simplicity and to illustrate only the essential features. Issues regarding removal of the cache aspect and knowing which methods need monitoring have been excluded. Also, issues regarding the cache concern have been omitted such as the cache replacement policy and altering the cache size to suit the conditions. However, the example does

illustrate how the required properties are implementable using a dynamic AOP framework, as will be discussed below.

5 Analysis

In the previous sections we have described some of the reasons why dynamic AOP is suitable for implementing an autonomic system. In this section we shall analyse the caching example to show the benefits of using dynamic AOP and then discuss some of the difficulties which could arise from using dynamic AOP. The problems described here will have to be overcome for an autonomic system to be successfully implemented using dynamic AOP.

5.1 Caching Example

In section 3 we listed a set of properties that make dynamic AOP suitable for implementing an autonomic system:

- Apply adaptations dynamically
- Easily remove adaptations
- Encapsulate adaptations
- Specify relationships
- Implement fine grained changes
- Apply adaptations to various points in a system

As can be seen from line 8 in the monitor advice code, an entirely new aspect can be introduced dynamically to the system. Although not used in the caching example the aspect could just as easily be removed, thus returning the system to its original state and allow the easy removal of adaptations using the following instruction:

```
Jac.remoteUnweaveAspect("CacheTest","s0","CacheAC");
```

Figure 6. Aspect Removal

It can be clearly seen that the adaptation to introduce the cache to the system is cleanly encapsulated. No external references are required to the base-code and no changes are required to the original code for the cache to work. This autonomic cache could be introduced to any other application with only changes being made to the pointcuts and the variable types stored in the cache. Although a simple example this demonstrates the encapsulation of adaptations and highlights the potential issues with reuse when needing to apply the same aspect to a variety of systems.

The example shows that fine-grained changes at the method level can be made. Not many applications will require changes at finer level of granularity than this. Although this caching example only shows changes at one point of execution it is clear that changes could be made at any point in the system.

The only property not demonstrated in the example is the definition of relationships between adaptations. This property was not shown in the example as only one adaptation was being made and so could not be related to anything else.

5.2 Issues Raised

This section will highlight some of the potential issues which could occur when implementing an autonomic system on a larger scale than the caching example.

Aspect Consistency

The crosscutting nature of aspects will mean that a single aspect could be applied to a number of different objects at a time. Each of these objects that have the aspect woven through it is likely to be in different states of 'use'. When the aspect needs to be removed from the objects this could be difficult due to the objects being in different states. It may be a requirement that for the aspect to be removed in a single operation the objects are all placed in a safe-state; this may not be possible.

If this problem is not considered and the aspects are removed regardless of the state, this could result in aspect inconsistency. This would mean that some objects that were in use could have the aspect still woven through it after the aspect has been requested to be removed while other objects will have had the aspect successfully removed.

When to Refactor Code and Disappearing Aspects

With aspects being added and removed arbitrarily, the system will no doubt at some stage need refactoring. The particular time chosen when this should be performed will need to be carefully selected as to avoid disruption.

Additionally, due to the fact that the majority of dynamic AOP languages do not alter the byte-code or source code in any way, dynamic aspects can be viewed as being temporary or volatile changes, in that once the application is stopped and restarted the system will be back in its original state with no aspects woven. However, a subset of the aspects that have been applied to an application may have to be made 'permanent' so that they are woven to the application whenever it is run.

Aspects of Aspects

Due to the dynamic nature of autonomic systems changes may need to be applied that were not anticipated while the system was being developed. This may also be true for aspects that have already been woven to the system. To use AOP to apply changes to the aspects already woven, the AOP language needs to allow aspects of aspects. However, the majority of dynamic AOP languages do not support this so some other method needs to be found; some static languages do support this such as Hyper/J, DJ and Aspect Collaborations.

Synchronising Aspects and Base-Code

When implementing a system which is able to modify its behaviour it is important to be able to predict how the system will behave once a change has been applied to it. So it is important to know when a new concern, which has just been woven, will be first executed. For example if a method is being executed and an aspect is woven around it, will the after part of the advice be executed? This is a fundamental feature of AOP that needs to be examined as the integrity of the application could be affected if the system behaves in unintended ways.

Describing Aspect Behaviour

Describing what each aspect will do is probably one of the more important properties/goals that needs to be implemented. The data which describes what the aspect does will be used when deciding which aspect needs to be used to alter the system behaviour in the desired way. There are various additional pieces of information that will also have to be stored such as dependencies, precedences, requirements, and other prerequisites.

This information relating to aspect behaviour and requirements could be stored in a meta-layer. This meta-layer could then be used by a change management process which would be able to determine whether the requirements set out in the meta-layer are currently met by the system and whether applying the aspect will affect the system stability or compatibility. Additionally, the meta-layer could support queries of the system regarding how the system has been altered to

allow adaptations to be removed when necessary. As well as inspection the meta-layer could also support adaptation of aspect information, to update the requirements as more information becomes available.

Aspect behaviour is one component of system behaviour that needs to be described as well as rules regarding when adaptations should occur being specified. Existing approaches have used static rules, modifying these rules dynamically would be one way to implement an autonomic system that can learn about itself. Again using a meta-level, the rules could be both inspected and adapted at run-time.

Security

The weaving process needs to be secure to prevent aspects being altered or aspects being introduced from an unverified source, since both of these situations could result in the system being left in an insecure state. Therefore some kind of authentication and encryption will be needed to be able to verify the source and to prevent changes being made to the aspects.

Atomic Weaving

Atomic weaving is also important to ensure that an aspect has been successfully woven. If it cannot be determined whether an aspect was fully woven it will be difficult later when trying to determine whether or not other aspects can be woven.

Suppose for example aspect A and aspect B are incompatible with each other. Aspect A is required to be woven with the base-code but it fails mid-way through the weaving process. It is now difficult to determine whether aspect B can be woven when it is needed. If atomic weaving was used and an aspect failed to be completely woven then the system would roll-back to the state it was in before the aspect was woven, this would then leave the system in a definite state.

Combining atomic weaving with some method of monitoring the aspects that have been woven/removed will be very useful when implementing an autonomic system as the current state of the system can be easily predicted. This is required when determining the compatibility issues between aspects.

Reuse

Being able to create a generic aspect which can be applied to a variety of systems and still maintain system consistency is vital when developing an autonomic system when using dynamic AOP. Although a set of systems may have similarities in their implementation, the specific implementation details will be different. Therefore it will be necessary for these aspects to be customisable; the Framed Aspect solution proposed in [13] may be suitable.

6 Other Techniques

The following section examines other techniques that can be used to compliment AOP in developing such a system, and will also look at other attempts to implement autonomous systems using AOP related techniques.

6.1 Complementary Techniques

HotSwap

HotSwap [9] is a new feature added to Java which allows entire classes to be swapped while the application which uses them is still running. When a replacement command is issued any methods of the old class that are still executing are allowed to finish but any new method calls are directed to the new class. The way HotSwapping would compliment AOP when implementing an autonomic system is obvious; instead of implementing complex aspects to substitute class behaviour, HotSwapping could be used. There is one limitation with using HotSwap; the old and

new classes must have the same interfaces when using JDK version 1.4. However, this limitation could be removed in later JVM's to allow more complex changes to be made.

Remote Debugging

With the wide use of distributed systems it is likely that the autonomic system will have distributed elements. Also, due to the nature of distributed systems they exist in an environment which is highly susceptible to change, a situation which is suited to autonomic systems.

Implementing an autonomic system in a distributed environment will need some sort of protocol to issue commands to remote components. The Java Platform Debugger Architecture (JPDA) [11] can be used to achieve this, as it allows remote JVMs to be debugged using the Java Debugger Wire Protocol (JDWP). The Java debugger has many useful features to introspect a running application which can be used for gathering information when deciding what actions need to be taken by the autonomic system. Additionally, the remote debugger allows HotSwapping, so remote classes can be swapped dynamically – another useful feature.

Dynamic Reconfiguration Algorithms

Dynamic reconfigurations have been made to component-based systems for sometime and so algorithms designed to aid the reconfigurations have been developed. The purpose of these algorithms is to examine the proposed adaptation and determine which components need to be directed to a safe state for the adaptation to take place, more information on this can be found in [7]. Elements of these algorithms should also be applicable to dynamically adding, removing and replacing aspects.

6.2 Other Work Using AOP Concepts

Previous work has been done by Yang [19] to implement a system which performs dynamic adaptation using AOP. This approach uses joinpoints to specify *where* the adaptation should take place, and a set of rules to specify the conditions *when* an adaptation should occur. Using rules is a static solution for self-configuring systems; the system should be able to learn the normal operating conditions and be able to identify when adaptations need to be made. Furthermore, although a small adaptive system that uses rules may be feasible, if a large system is being implemented, it would be nearly impossible to specify all the rules to define all the adaptations that may be necessary.

Another solution to achieve adaptation in applications using AspectJ is proposed by Dantas et al [3]. A number of key components have been identified to implement this solution:

- Base Application
- Adaptability Aspects
- Auxiliary Classes
- Context Manager
- Adaptation Data Provider

The base application contains the core application code without any code to implement adaptations. The adaptability and auxiliary classes co-operate together to implement the adaptations, the aspects specify how the application should be changed and then some tasks are delegated to the auxiliary classes. The context manager is responsible for monitoring the application and triggering any adaptations implemented in the aspects. Finally the adaptation data provider is a set of classes which provide context data to the dynamic adaptations, the same context change could lead to a different adaptation occurring if different data is passed from this module.

The developers claim the architecture pattern is dynamic in that adaptations can be made at run-time and the adaptation data provider can alter the application adaptations. However, it is not possible to add new adaptations at run-time as AspectJ is being used, which is the main limiting factor of this implementation.

An adaptation framework developed by Pierre-Charles David et al [4] is implemented using the Fractal component model. Although this framework does not use AOP directly it still uses the main fundamental concept – separation of concerns. The creators believe that the adaptation process should be kept separate from the main business logic of the application. This is something that we also aim to achieve by using AOP directly; we hope to be able to use our developed framework to automate a variety of systems, and so need to separate the adaptation process from the application.

The Fractal component model uses a controller which all communication passes through; the developers have modified this controller so that additional code can be inserted at the appropriate time and so can alter the application behaviour. This is similar to our approach, but instead we will be using dynamic AOP to insert this additional code.

Prose has been used to implement a system which possessed a certain degree of autonomous behaviour. MIDAS [17] was added to Prose to implement an extension management system. MIDAS was used to control the distribution of aspects in a mobile environment. The given potential scenario for use was in a factory setting, where machines could be moved to different locations. Whenever a machine entered a new location an extension base distributes the aspects to the nodes that contains an extension receiver to allow receive and weave these aspects. The aspects allow the machines to operate in the ways desirable to that particular location. Although this could be classed as an autonomic system, it is quite limited and only suited to a distributed environment. The solution we propose would be able to make more decisions about how to behave and would be suited to a variety of different applications.

7 Conclusion and Future Work

This paper has outlined the case for using dynamic AOP to implement an autonomic system. Both the benefits and problems have been discussed. There are clear advantages in using dynamic AOP for such a system which are illustrated using a simple cache example. Although there are many benefits from using dynamic AOP, several issues are raised which are mainly related to maintaining system stability. None of the mentioned issues have yet to be fully solved by the AOP community; however some of these problems are present in other related work in dynamic reconfiguration. We will have to find a solution which is suited to dynamic AOP.

The goals and future work of this project will aim to solve the outstanding issues listed in section 5. Although these goals are focused upon aiding the implementation of an autonomic system, the issues mentioned will be present in any application created using dynamic AOP; this work will hopefully contribute to both the autonomic and AOP communities.

References

[1] AspectWerkz, “AspectWerkz Overview”, http://aspectwerkz.codehaus.org/, 2003.

[2] BCEL, “BCEL Home Page”, http://jakarta.apache.org/bcel/, 2004.

[3] Dantas, A., Borba, P. “Adaptability Aspects: An Architectural Pattern for Structuring Adaptive Applications with Aspects”, Proceedings of SugarloafPLoP 2003 Conference, 2003.

[4] David, P., Ledoux, T., “Towards a Framework for Self-Adaptive Component-Based Applications”, Proceedings of FMOODS/DAIS 2003, 2003.

[5] Gell-Mann, M., “What is complexity?”, Complexity Vol. 1 No. 1, 1995.

[6] Hicks, M., Moore, J. T., Nettles, S., “Dynamic Software Updating”, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001.

[7] Hillman, J., “Dynamic Adaptation of Dependable Systems”, 8th CaberNet Radicals Workshop, 2003.

[8] Horn, P., “Autonomic Computing: IBM’s Perspective on the State of Information Technology”, 2003.

[9] HotSwap, “HotSwap Project Research Publications”, http://developers.sun.com, 2004.

[10] Hughes, D., Greenwood, P., Blair, L., “Aspect Testing Framework”, FMOODS/DAIS Student Workshop, 2003.

[11] JPDA, “Java Platform Debugger Architecture (JPDA)”, http://java.sun.com/products/jpda/, 2004.

[12] Kiczales, G. et al, “An Overview of AspectJ”, Proceedings of ECOOP pp 327-353, 2001.

[13] Loughran, N., Rashid, A., “Supporting Evolution in Software using Frame Technology and Aspect-Orientation”, Workshop on Software Variability Management, 2003.

[14] Pawlak, R. et al, “JAC: A Flexible Solution for Aspect-Oriented Programming in Java”, Reflection 2001, 2001.

[15] Pawlak, R. et al, “Dynamic Wrappers: Handling the Composition Issues with JAC”, TOOLS USA 2001, 2001.

[16] Popovici, A., Gross, T., Alonso, G., “Dynamic Weaving for Aspect-Oriented Programming”, AOSD 2002, 2002.

[17] Popovici, A., Frei, A., Alonso, G., “A Dynamic Middleware Platform for Mobile Computing”, Middleware 2003, 2003.

[18] Seinturier, L. et al, “JAC Milestones 2001”, Research Report LIP6 2001/025, 2001.

[19] Yang, Z., “An Aspect-Oriented Approach to Dynamic Adaptation”, WOSS 2002, 2002.

Using Dynamic Aspects in Music Composition Systems

Patrick Hill
Simon Holland
Robin C. Laney
The Open University, Walton Hall
Milton Keynes. MK7 6AA
PatrickHill@bcs.org.uk
s.holland@open.ac.uk
r.c.laney@open.ac.uk

Abstract

Aspect-oriented programming (AOP) attempts to modularise crosscutting concerns in software. Initial approaches to AOP have used static weaving techniques in which crosscutting implementation, encapsulated by aspects, is merged or woven into base code at compile-time. Research into dynamic aspects suggests various ways in which crosscutting implementations may be dynamically woven into code, enabling aspects to be defined and composed at run-time.

It has been suggested, in [15], that AOP might be usefully applied at the end-user level in applications that support multidimensional creative processes, and in particular, of music composition. In this paper we extend this argument to suggest that dynamic aspects are essential to this application. We motivate our argument with a high-level description of crosscutting that exists within music composition, and ways in which these crosscutting concerns, and requirements for their management, have arisen from our initial use of static aspects in music composition. We then evaluate some of the ways in which current research into dynamic aspects might be utilised in addressing these requirements.

1 Introduction

Aspect-oriented programming (AOP) is a technique that aims to assist software developers in the separation and composition of various dimensions of concern across a range of software engineering tasks. Early compositional AOP tools, such as AspectJ, operate by statically composing, or *weaving,* crosscutting concerns, expressed as *aspects*, with basic concern code, expressed as *classes*. However, static weaving, by definition, assumes that aspectual relationships are largely invariant and that they can be determined at design-time and are therefore tied to a particular class-graph [20].

In contrast, dynamic aspects variously offer the potential to defer binding of particular aspect implementations until run time. Unlike the statically woven aspects of AspectJ, dynamic aspects persist at run-time and it therefore becomes possible to dynamically modify aspectual relationships, enabling aspects to be added, withdrawn, or replaced depending upon dynamic context.

In [15] we argue that music composition can be viewed in terms of composition of various dimensions of musical concern and that analogies exist with AOP. We suggest that AOP may be used at the *end-user* level in systems that support music composition. In this paper we further suggest that aspectual relationships in musical composition are largely dynamic, and that dynamic

aspects could prove a useful technology in the development of aspect-oriented music composition tools.

2 Separation and Composition of Concerns in Music

Music composition is a creative process in which the separation and composition of dimensions of concern are important and pervasive problems. Multidimensional tangling and scattering exists not only within the structure, representation and manipulation of musical data, but also in the cognitive processes of composition [15].

It is common experience that music is not merely a random stream of sound events. Rather, the composer typically works with a limited set of musical resources that are manipulated, in various ways, to form a logical and coherent whole [30]. The 'musical surface' of a musical composition, which is perceived by the listener in terms of *pitch*, *duration*, *loudness* and *timbre*[1] [21], can be viewed as the result of the composer's weaving together of a 'tangled web' of musical structures and dimensions [9].

Although the processes of software engineering and music composition are clearly different, there are some parallels between the two. We can, for example, draw a broad analogy between the notation that is traditionally output as part of a composition process and the set of instructions executed by a computer as the result of a software engineering process. In both cases, the outputs are, largely, sets of low-level performance instructions resulting from the composition of high-level abstractions.

For example, consider the musical gesture of *crescendo*, ie. 'getting louder over time'. A crescendo might be readily achieved by simply increasing the value of the 'loudness dimension', eg. by striking piano keys with more force or blowing a trumpet more forcefully[2]. However, there are other dimensions that may also be modified in order to obtain a crescendo effect. For example, the composer might choose to

- introduce additional instruments (timbre),
- use different pitches (pitch),
- modify the arrangement of harmonies (pitch /timbre)

and so forth.

Thus the basic 'crescendo' concern may be scattered among other musical dimensions. Moreover, the particular crescendo implementation used might depend upon musical context. For example, while some sections of a musical piece might be written for a full orchestra, other sections might be written for strings only. Thus the 'introduction of additional instruments' approach to crescendo might itself be limited in the instruments that may be used.

We choose this example as one that is readily understood and that does not mandate in-depth discussion of musical technicalities. There are numerous other documented instances of crosscutting in music. Examples include separation of metre and melody [18], the impact of tempo on performance [11], and the interrelationship between orchestration and composition [26]

While software is typically composed through the use of automated tools; compilers, configuration management etc, even with computer assistance, the music composer is often forced to express high-level musical ideas in terms of tangled, low-level musical detail, by manually weaving together various dimensions. The requirement to express music in such a detailed 'note-list' representation rather than as higher-level constructs is incompatible with the

[1] Timbre describes those qualities of a sound that enable the listener, for example, to distinguish between a trumpet and a violin.

[2] In practice, these techniques would also affect timbre.

creative process itself [22]. Moreover, musical composition does not appear to be a linear process. Rather, composers tend to sketch out and elaborate ideas iteratively and across multiple, possibly incomplete, dimensions [31,25,32]. During the composition process, certain musical elements may be created which the composer wishes to preserve and use, but not necessarily in the present composition [32]

From the viewpoint of the music analyst, just as it would be difficult to identify aspects from reverse-engineered bytecode produced by AspectJ, musical intent is often obscured in the musical score produced by the composer. As Raes [29] points out, musical scoring systems, conventional or otherwise, do not express the 'conception' or 'flow of ideas' within a musical composition. This is not to underestimate the power of algorithmic musical composition systems but note that here too, musical dimensions are often scattered or tangled.

We believe that AOP techniques might be usefully applied to the domain of computer assisted music composition as a way to weave together separately described musical elements that express musical *intent*. An AOP-based music creation environment could enable the composer to work in an iterative experimental fashion that supports the creative process.

3 Why are Dynamic Aspects Important to Musical Composition Applications?

Our initial research has shown that statically composed aspects, of the type implemented by AspectJ, may be used, with some success, to help separate musical concerns and compose them into a musical piece. For example, we have used AspectJ to construct the first few bars of Widor's famous organ Toccata, using a core program that represents a sequence of chords, and aspects that implement crosscutting concerns, such as changes of key, temporal position and duration of chords, and transformation of the chords into the left-hand and right-hand parts of the original score. In considering the extension of this system to generate the entire piece, it was observed that some general requirements could not be met by AspectJ.

- Music is often based on the variation and juxtaposition of a small number of musical elements [30]. From an AOP perspective, this means that aspectual relationships do not necessarily persist for the entire duration of a musical piece. This is in some ways analogous to the 'Jumping Aspects' problem [5], in that a particular aspect behaviour might be desirable only within certain musical contexts. Gybels [13] observes that some crosscut languages, particularly that used in AspectJ, are not Turing Complete, and are therefore unable to evaluate dynamic expressions. Clearly, 'enabling conditions' based on dynamic context might be encoded into the advice, but this could lead to over-complicated code and unclear separation of concerns between pointcut and advice.

- For example, consider the case where within the same piece of music some crescendi are realised by additional instrumentation, while others are realised by a simple increase of 'loudness'. A 'pointcut' on a crescendo would require different 'advice' depending on context.

- Dynamically installable aspects might be used to separate concerns such that the selection of 'which' crescendo implementation is used is described separately from the 'aspect' that invokes the selected crescendo implementation at crescendo pointcuts.

- By definition, static aspects cannot be defined and applied interactively at run-time. Music composition is a creative art that involves experimentation and iteration [25,31,32]. The development of an interactive music composition system that enables the composer to selectively define, apply, refine, and withdraw aspectual relationships presupposes a dynamic aspect platform.

- Given such an interactive music composition system, it is possible that certain aspects could be constructed that may have application in a range of musical compositions, not only the one in which they were defined. In a similar way to the 'buy-don't-build' [7] methodology espoused by proponents of component based software development, the ability to encapsulate musical aspects as components that may be subsequently 'plugged-in' and reused in other musical composition projects is attractive.
- Greater separation of concerns might be achieved with the facility to compose aspects with other aspects. While this does not necessarily require dynamic aspects, it is not currently possible using, for example, AspectJ.

4 Dynamic Aspect Systems

In this section we overview ways in which current dynamic aspect systems might help in the development of a musical composition system as outlined above.

4.1 Event-Based Dynamic AOP

Whereas systems such as AspectJ determine pointcuts from source-code inspection, event-based systems, such as EAOP [12], Axon [1] and PROSE [28] utilise various techniques to generate events at runtime. These events, which function as joinpoints, are intercepted and used to invoke separately defined crosscutting implementations.

EAOP performs source-code modification to produce a framework that instruments the source application code such that execution events are raised to an *event monitor*. Aspects, which are coded as pieces of Java code, are invoked by the event monitor upon receipt of particular events. An 'aspect tree', which specifies how events are routed to aspect code, is maintained by the monitor. Dynamic AOP is achieved by the ability to modify the aspect tree at run-time. Aspects may be composed with other aspects through the use of EAOP's composition operators.

In contrast, both Axon and PROSE utilise the services of the Java debugger interface (JVMDI) to raise events at appropriate points, such as method calls, in the programs execution. This approach does not require source-code modification. Axon uses an API to programmatically define aspects in terms of dynamic associations between pointcuts and advisory units. Advisory units, analogous to AspectJ's advice, are written as plain Java classes. In PROSE, Java classes representing aspects are defined as subclasses of the PROSE class Aspect. As such, each aspect class contains one or more Crosscut objects, each of which equates to the combination of an AspectJ pointcut **and** an advice. Pointcuts are defined using *specializers*, which are specified using an AspectJ-like pointcut syntax. Aspects may be added or removed through interaction with PROSE's ExtensionManager interface. An interesting feature of this approach is the ability to manipulate aspect instances from outside of the JVM in which the application is running.

Event-based AOP has an immediate appeal for musical applications, since parallels can be drawn with the event-based nature of the industry-standard Musical Instruments Digital Interface (MIDI) model that is used by many musical software systems and the general observation that music is perceived as discrete audio events in time. Advantages of event-based AOP include the clear separation of aspect and application code. Axon goes further and separates pointcuts from advice. However, the event generation systems employed by the three systems overviewed above are not ideal. Typically, events are 'over eager'; they are generated irrespective of whether they are required by aspects. EAOP's source code modification conflicts with a general AOP requirement of non-invasiveness [1] but the use of the JVMDI imposes a performance overhead that might render it unsuitable for realtime applications.

4.2 Meta Programming & Reflection

Metaprogramming and reflection techniques, in principle, enable a piece of software to both discover and modify its internal structure. Thus metaprogramming enables programs to reason about themselves. To do this, a programming language or environment must expose internal structure such that it can be programmatically manipulated as data, through the process of *reification*. Meta Object Protocols (MOPs) enable structural program elements to be manipulated as object-oriented encapsulations of reified data and appropriate methods. Indeed, AOP has its roots in MOP research [33], indeed AOP itself is a computational reflection mechanism [17].

Meta Programming has been used as an underlying technology that enables AOP. Examples of such enabling technologies include MethodWrappers [4], AOP/ST [6], and Handi-Wrap [2], all of which support the dynamic composition of method code with additional 'wrapper' code that might constitute advice.

AspectS [16] utilises MethodWrappers [4] to implement a dynamic AOP system for Squeak Smalltalk. AspectS defines a set of base classes from which aspects are derived. Aspects are themselves written in Smalltalk, and are dynamically woven or unwoven, by automatically modifying the relevant Smalltalk Class descriptions (as described by the aspect's joinpoint) such that subsequent calls to the original method are redirected to a new wrapped method. The wrapped method, which is dynamically compiled into the Smalltalk system, invokes the aspect's advice and the original method in ways that are analogous to AspectJ's before(), after() and around() advices. It is noted however [17] that AspectS focuses on message passing, and that other types of joinpoint, such as member variable access, are not easily achieved through the use of Smalltalk's MOP.

Gybels [13] proposes the use of a Logic Meta Programming Language as an Aspect language. His "Andrew" system uses the Smalltalk Open Unification Language (SOUL) to implement a dynamic aspect system with an aspect model similar to that of AspectJ. SOUL is a variant of PROLOG, but its implementation enables dynamic interaction between itself and Smalltalk. For example, SOUL facts may contain arbitrary dynamic Smalltalk expressions and the result of these expressions may be bound to SOUL logic variables. Andrew implements pointcuts in terms of SOUL predicates that relate to typical AOP joinpoints such as method invocation, methods reception, member variable assignment and so on. Aspects are implemented as the combination of pointcuts and advice. Although Andrew implements before() and after() advice, there is currently no support for around() advice and as such it is currently not possible for aspects to choose not to execute methods, as they can in AspectJ. The use of a logic language for aspect, and particularly pointcut, definition is both intuitive and flexible. This is enhanced by the language symbiosis between SOUL and Smalltalk.

Metaprogramming based AOP is heavily dependent on the reflective nature of the underlying language, which may explain why many such approaches target the highly reflective Smalltalk language. A Java equivalent of AspectS, for example, would not be possible because Java's reflection capability is limited to introspection and does not permit intercession [17]. Nevertheless, Java-based dynamic AOP systems that utilise metaprogramming do exist. A hybrid solution for Java, which uses static weaving and a reflective Java environment is presented in [10], while Handi-Wrap [2] uses Java extensions (implemented using Maya [3]) to describe dynamic wrappers that are realised using compile-time reflection.

Metaprogramming appears to be a key enabler of run-time dynamic AOP systems, indeed certain MOP based AOP implementations may be considered as disciplined metaprogramming [17]. We also note that, many musical research systems utilise highly reflective languages; key examples include. Open Music [35] and Symbolic Composer [36] written in LISP, and MODE [27] and DMix [23] written in Smalltalk.

In the context of music composition systems, one approach might therefore be to synthesise a dynamic AOP music composition system from existing music systems, such as MODE, in

combination with a dynamic AOP system such as AspectS or Andrew. We can also imagine scenarios where introspection into the musical structure itself might be valuable. Consider the example of 'stretching' the length of a section of music. This operation, termed *augmentation* [30], typically involves multiplying the onset time and duration of sound events by some factor. Thus augmenting four consecutive notes each of 1 second duration by a factor of two yields four consecutive notes each of 2 seconds duration. However, in the case where the music being augmented is, for example, a 'drum roll', then instead of multiplying durations, it is necessary to preserve durations, but add additional notes to fill the augmented duration. Thus augmenting a 'drum roll' of four consecutive notes each of 1 second duration yields a sequence of 8 consecutive notes, each of one second duration. Using introspection, an 'augmentation' aspect could identify the kind of musical structure that was being augmented and invoke the correct behaviour. This *behavioural abstraction* is one of the motivating factors behind the LISP-based Nyquist [37] music system.

Like the use of JVMDI, however, MOP and reflection typically impose a performance overhead [17], which may make them unsuitable for realtime applications.

4.3 Aspectual Components

In principle, it is possible to design aspects that have a more general application than the specific application in which they are first defined. However, aspect systems such as AspectJ and AspectS require that information relating to the static class structure of an application be encoded into aspect definitions. In AspectJ, for example, pointcuts must refer to specific class and method names. Thus aspects may only be re-used in applications that include a class subgraph that matches that referenced by the pointcut definitions. Further, it has been observed that undisciplined construction of aspects in AspectJ prevents aspects from being extended and reused [14].

In the spirit of structure-shy Adaptive Programming [19], Aspectual Components [20] permit aspect binding to be deferred by introducing a level of abstraction that divorces the aspect definition from a particular class structure or method protocol. This is achieved by defining an aspect, termed a *component*, in terms of its own class structure or Participant Graph (PG) that represents an abstract slice through a set of possible concrete class graphs (CGs). Aspects are subsequently bound to a CG using *connectors* that map both classes and methods to the PG. Thus aspects may be defined as discrete crosscutting components that may be applied to any given application class graph through separate connector specifications. It is unclear, however, how ACs can be made to handle dynamic context, and thus avoid Jumping [5] or Vanishing [8] aspects.

A practical implementation of ACs is provided by the JAsCo system [34]. In JAsCo, the standard java component metaphor, JavaBeans, is extended to form *aspect beans*. Aspect beans correspond to the component structure of ACs and encapsulate the behavioural properties of an aspect, providing an abstract interface through which these behaviours may be invoked. Like ACs, JAsCo utilises the concept of a *connector* to establish a relationship between aspect behaviour and a concrete class graph. JAsCo also supports the dynamic insertion and removal of aspect beans.

Aspectual Components form the basis of technologies that enable aspects to be defined as 'plug-ins' across a range of applications and as such, promote re-use. In a musical context, the use of Aspectual Components would permit the definition of a range of crosscutting musical concerns that could then be applied to multiple composition projects.

The Caesar system [24] further abstracts the definition of an aspect. In Caesar, aspects may be abstractly described as a set of *required* and *expected* interface implementations, a so-called Aspect Collaboration Interface (ACI). Required interfaces are implemented by the *binding* of an aspect to concrete classes. For example, if an aspect requires additional methods to be

implemented in a base class, then these methods are implemented in the binding. The aspect implementation itself implements the ACI's expected interfaces. The binding and a concrete aspect implementation are then woven together to form a *weavlet*. This approach makes it possible to define and use various aspect implementations without necessarily redefining the binding and, conversely, it is possible to reuse aspect implementations with bindings to other concrete classes.

Like JAsCo, Caesar supports dynamic deployment of aspects. Caesar also supports the concept of *aspectual polymorphism*, permitting aspects to be deployed whose type is not known until run-time. Another interesting feature of Caesar is that dynamic deployment affects only the control flow of the deployment block. Other threads, even those executing the same code, are not affected.

Dynamic deployment is clearly important to systems, like interactive music composition applications, in which aspects are likely to be selected and activated at run-time, such as by the user selecting an aspect from a menu. The aspect deployment mechanism used by Caesar, which affects only the control flow of the dynamic deployment statement block, could be of value in a musical context where aspects are to be applied to specific instances of a repeating musical entity.

5 Conclusions

In this paper we have outlined that music composition is a domain in which multidimensional scattering and tangling is very much evident. We explained that the traditional role of the music composer is to manually weave together these dimensions to form the musical surface that is perceived by listeners.

We have suggested that aspects could be used to help in the construction of musical composition systems that enable composers to express musical intent, and that perform low-level weaving of musical data based upon higher level musical descriptions. We note, however, that the relationships between musical dimensions, even over common high-level concepts, such as crescendo, are not static, and depend both upon the composer's wishes and musical context. We believe that dynamic aspects offer a way to manage these dynamic crosscutting concerns in the provision of AOP-based interactive music composition tools.

We have briefly outlined some of the current dynamic aspect technologies and indicated their various strengths and weaknesses and possible uses in relation to musical composition applications.

Our future research will consider ways in which dynamic aspects may be implemented and used in the development of an interactive computer based music composition system. Key areas of interest are the development of a dynamic aspects system that supports the requirements of music composition, in terms of the modelling and implementation of dynamic musical relationships, and the extension of the AOP paradigm to the user-level. In particular we wish the user to be able to interactively and dynamically define, apply, and modify crosscutting relationships and to store them for future application in other musical composition projects. As an interactive system, we require dynamic aspects that are responsive, and with the potential for them to support music generation in realtime. As an end-user application, we also require stability and simplicity.

References

[1] Aussmann, S., Haupt, M. Axon – Dynamic AOP through Runtime Inspection and Monitoring. ASARTI Workshop 2003.

[2] Baker, J., Hsieh, W. *Runtime Aspect Weaving Through Metaprogramming*. AOSD 2002.

[3] Baker, J., Hsieh, W. C. *Maya: Multiple-Dispatch Syntax Extension in Java*. In Communications of the ACM. 2002.

[4] Brant, J., Foote, B., Johnson, R. E., Roberts, D. *Wrappers to the Rescue*. ECOOP 1998.

[5] Brichau J., De Meuter W., De Volker K. *Jumping Aspects.* Workshop of Aspects and Dimensions of Concerns ECOOP. 2000.

[6] Bollert, K. *On Weaving Aspects*. In Proceedings of Aspect-Oriented Programming Workshop at ECOOP 1999.

[7] Brooks, F. P. *No Silver Bullet: Essence and Accidents of Software Engineering*. Computer, vol 20, no. 4. 1987

[8] Constanza. P. *Vanishing Aspects*. OOPSLA 2000

[9] Dannenberg, R. B., Desain, P., Honing, H. *Programming Language Design for Music*. In G. De Poli, A. Picialli, S. T. Pope, & C. Roads (eds.), Musical Signal Processing. 271-315. Lisse: Swets & Zeitlinger. 1997.

[10] David, P-C., Ledoux, T., Bouraqadi-Saâdani, N.M.N. *Two-Step Weaving with Reflection using AspectJ*.

[11] Desain, P., Honing, H. *Tempo Curves Considered Harmful*. Contemporary Music Review. 7(2). 1993.

[12] Douence, R., Sudholt, M. *A model and a tool for Event-based Aspect-Oriented Programming (EAOP)* . TR 02/l 1/INFO, lcole des Mines de Nantes, french version accepted at LMO'03, 2nd edition, Dec. 2002

[13] Gybels, K. *Aspect-Oriented Programming using a Logic Meta Programming Language to express cross-cutting through a dynamic joinpoint structure*. Ph.D. Thesis 2001.

[14] Hannenberg, S., Unland, R. *Using and Reusing Aspects in AspectJ*. OOPSLA 2001

[15] Hill, P., Holland, S., Laney, R. C. *Using Aspects to Help Composers*. Technical Report TR 2003/21. Open University Dept of Computing 2003.

[16] Hirschfeld, R. *Aspect-Oriented Programming with AspectS*. 2002

[17] Kojarski, S., Lieberherr, K., Lorenz, D. H., Hirschfeld, R. *Aspectual Reflection*. AOSD SPLAT Workshop. 2003.

[18] Lerdahl, F., Jackendoff, R. *A Generative Theory of Tonal Music*, MIT Press, 1983.

[19] Lieberherr, K. J., Silva-Lepe I., Xiao C. *Adaptive Object-Oriented Programming using Graph Customisation*. College of Computer Science, Northeastern University, 1994.

[20] Lieberherr, K., Lorenz, D. and Mezini, M. *Programming with Aspectual Components*. Technical Report, NU-CCS-99-01, March 1999.

[21] Loy, G., Abbott, C. *Programming Languages for Computer Music Synthesis, Performance and Composition*. ACM Computing Surveys, Vol.17, No. 2. June 1985

[22] Oppenheim, D.V. *Towards a Better Software-Design for Supporting Creative Musical Activity*. ICMC 1991.

[23] Oppenheim, D. *DMIX: A multi faceted environment for composing and performing computer music*. Mathematics and Computers, 1996.

[24] Mezini, M., Ostermann. K. *Conquering Aspects with Caesar*. AOSD 2003.

[25] Pearce, M., Wiggins, G. A. *Aspects of a Cognitive Theory of Creativity in Musical Composition.* Dept of Computing, City University, London. 2002.

[26] Piston, W. *Orchestration,* Gollancz 1961.

[27] Pope, S. *Introduction to MODE: The Musical Object Development Environment*. In The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology, S. T. Pope, ed. MIT Press. 1991.

[28] Popovici, A., Gross, T., Alonso, G. *Dynamic weaving for aspect oriented programming*. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development, April 2002

[29] Raes, W-G. "*The multitasker as an approach in musical composition*" 1997 http://logosfoundation.org/g_texts/multitaskers.html

[30] Shoenberg, A. (Strang F, Stein L. eds). *Fundamentals of Music composition*, Faber and Faber. 1967.

[31] Sloboda, J. A. *The Musical Mind. The Cognitive Psychology of Music*. Oxford University Press. 1985.

[32] Speigel, L. *Old Fashioned Composing from the Inside Out: On Sounding Un-Digital on the Compositional Level*. Proceedings of the 8th Symposium on Small Computers in the Arts, Nov. 1988.

[33] Sullivan, G. T. *Aspect-Oriented Programming using Reflection*. OOPSLA 2001.

[34] Suvée, D., Vanderperren., W., Jonckers., V. *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development*. AOSD 2003.

[35] http://www.ircam.fr/produits/logiciels/openmusic-e.html

[36] http://www.mracpublishing.com/scom/

[37] http://www-2.cs.cmu.edu/~rbd/doc/nyquist/root.html

Using Dynamic Aspects to Distill Business Rules from Legacy Code

Isabel Michiels
Theo D'Hondt
PROG, Vrije Universiteit Brussel (VUB)
Kris De Schutter
Ghislain Hoffman
INTEC, Universiteit Gent (UGent)

Abstract

Large organizations often rely on business rules to express their business constraints and very often these rules are scattered throughout different parts of the source code. Although call-stack context and other dynamic events provide a valuable view on (old) code, checking why the output of their system produces certain values remains a complex and time-consuming process. In this paper we advocate the use of dynamic aspects to facilitate and optimize the process of distilling business rules from legacy code. We demonstrate this use through a possible scenario of investigation of a small but real life case study and conclude with our envisioned practical implementation.

1 Introduction

When information systems are created for use within (large) organizations, many business rules are embedded into the software as some kind of constraints. On first implementation of these business rules they are usually fragmented and inserted into many different parts of the source code, which makes it hard to localize them at a later stage when, for example, the software is evolving.

Information systems that have been around for a long time typically suffer from this scattering of business rules; although some rules have to be continuously adapted and are thus kept in human memory, there are other business rules that, once they are implemented, are left unchanged and, through the passage of time, are somehow "lost" in the source code. This gets even more problematic when documentation is not being kept up to date.

Checking whether the output of an information system is correct, or why it produces a certain value, therefore becomes very difficult. The only plausible approach to tackle this scattering is to trace program execution, which can be a complex and time-consuming process.

As an illustration, consider being confronted with the following situation: our accountancy department reports that several of our employees were accredited an unexpected and unexplained bonus of €500. Accountancy rightfully requests to know the reason for this unforeseen expense. Not knowing the exact cause, we are left faced with having to comprehend an old and poorly documented system.

2 Research Context

The problems described here were encountered several times within the ARRIBA[1] project, a generic research project funded by the IWT, Flanders[2], which started out in October 2002 and will last four years [2].

The main goal is to provide a methodology and its associated lightweight tools in order to support the integration of disparate business applications that have not necessarily been designed to coexist.

Inspiration for this project comes from two driving forces: on the one hand we have a consortium of research groups[3] that have been active in the field of software engineering and, more particularly, in re(verse) engineering, software evolution and software architectures. On the other hand, we have the recently created forum of Belgian enterprises[4] (large and small) interested in a joint initiative to identify generic problems and likewise generic solutions plaguing their ICT base.

The object of this investigation is therefore the identification of mainstream ICT problems within this forum of enterprises that rely on information technology for their critical business activities.

Part of this is covered by the newly named discipline of Enterprise Application Integration (EAI), and re(verse) engineering; another part lies in the use of AOSD techniques for code instrumentation as an important tool to aid program comprehension.

3 Difficulties encountered

With an estimated 60\% to 80\% of all business applications still written in COBOL[4], it was no surprise to find exactly this in the code base of the companies involved in ARRIBA. COBOL therefore quickly gained much of our focus.

Working with COBOL has its difficulties:

- The applications concerned are no longer understood. Major mission-critical applications were developed in the 70's by programmers that are no longer working at the company, or have moved on to other projects.
- The code is badly structured and poorly documented. The amount of code is huge (millions of LOC) and has been adapted many times for several reasons (switching platforms, year 2000 conversions, transition to the Euro currency,…). So keeping the documentation synchronized with those evolutionary changes didn't always happen.
- Logic is spread out over the entire application. COBOL has only limited modularity mechanisms. Therefore complex logic had to be manually distributed over the programs.
- COBOL as a programming language is no longer understood. Languages like COBOL are no longer very popular with the new generation of programmers, nor are they being actively taught to students. We are more and more faced with a new generation of computer scientists with a different kind of background.
- Specific COBOL language constructs: COBOL language constructs such as REDEFINE, GO TO or ALTER make it extra difficult to trace the execution of a program.

[1] http://arriba.vub.ac.be/

[2] http://www.iwt.be/

[3] Vrije Universiteit Brussel (VUB), Universiteit Gent (UG), Universiteit Antwerpen (UA); supported by UCL in Louvain-La-Neuve, and SCG, Berne, Switzerland.

[4] Inno.com, KBC, LCM, Banksys, Toyota, KAVA and Pefa

4 Proposed approach

To tackle some of the above problems we propose to use dynamic aspects as an aid for semi-automating the process of tracing the execution of a (COBOL) program.

Dynamic aspects are aspects that are used when you want to invoke some behavior based on the dynamics of program execution[1]. This technology will allow us to inspect specific parts of the dynamic execution of an application.

To elaborate on this point, let us reconsider the situation presented in the introduction: being faced with an unexpected bonus of €500 for some of our employees. We now present a simplified but realistic scenario of how dynamic aspects can help us get a handle on this problem.

4.1 Possible scenario of investigation

First thing we have to figure out is which variable or record holds the value for the bonus. Accountancy has provided us with printed reports showing the questionable results. We use these to find the routine which has generated these results through a simple search on strings, and find that the data (the bonus in euros) is being held in a variable named (for instance) BNS-EUR. (Because we plan on using it later we also write down the variable holding the employee id number.)

Looking into the definition of BNS-EUR, it tells us that it is defined as an edited picture[1]. We conclude that this variable is only used for pretty printing the output, and not for performing actual calculations. At some time during execution the correct value for the bonus was moved to BNS-EUR, and consequently printed. We now have to find what variable that was.

Rather than looking at all MOVEs to BNS-EUR, we will cut down the list to those MOVEs which occurred while processing one of the 'lucky' employees (which we can deduce from the reports we received from accountancy). This is where our first dynamic aspect helps us out. It limits the data we have to look at by allowing us to apply previously gained knowledge.

We find the possibilities to be one of several string literals (which we can therefore immediately disregard) and a variable named BNS-EOY (whose name suggests it holds the full value for the end-of-year bonus).

Our next step is to try to figure out how the end value was calculated. Knowing that would allow us to check the figures and maybe spot an error. To achieve this we set up another aspect to trace all statements modifying the variable BNS-EOY.

Consider the following piece of (pseudo) COBOL code to demonstrate some of the things we (might) have to capture in this phase:

```
03000       READ EMPLOYEE.
03100*      ...

04800       ADD B31241 TO BNS-EOY.
04900*      or
05000       COMPUTE BNS-EOY = BNS-EOY + B31241.
05100*      or
05200       MOVE B31241 TO BNS-EOY.
05300*      or...
```

[1] In COBOL you define a variable as being a picture of a number of characters or numeric values, like "A-VAR PIC 9(2)", which means that A-VAR can hold a numeric value of 0 up to 99. You can also edit these PICs by making them ready for displaying them the way you want, like a date for example: "A-DATE PIC 99/99/99". [6]

These include arithmetic statements and MOVEs. Dynamic aspects allow us to get this lifetime trace of a variable. And again we can limit these lifetime-traces to those which occur while processing specific employees.

In doing so we get lucky and find a variable (cryptically) named B31241, which is consistently valued €500, and is added to BNS-EUR in every trace.

Before moving on we'd like to make sure we're on the right track. We want to verify that this addition of B31241 is only triggered for our list of 'lucky' employees. Again, a dynamic aspect allows us to trace execution of exactly this addition and helps us verify that our basic assumption holds indeed.

Knowing what is added leaves us with the question of why. Unfortunately, the logic behind this seems spread out over the entire application. So to try to figure out this mess we would like to have an execution trace of each *lucky* employee, including a report of all tests made and passed, up to and including the point where B31241 is added. Dynamic aspects allow us to get these specific traces. Comparing these will narrow down our search and help us find our way inside the original code.

This is where our story ends. We find that B31241 is part of a business rule: it is a bonus an employee receives when he or she has sold at least 100 items of the product with number 31241. Apparently this product code had been assigned to a new product the year before. It once was associated to another product which had been discontinued for several years. The associated bonus was left behind in the code, and never triggered until employees started selling the new product.

4.2 Future Implementation

The way we want to implement dynamic aspects in COBOL is to use a declarative language at the meta level as a pointcut language that will reason statically about dynamic execution traces. A declarative language is especially suited for expressing constructs like mentioned in our COBOL example in section 4.1: for capturing the modification of a variable one can write a predicate which expresses the different ways for changing a variable. This way you create an abstraction layer which makes it easy to adapt the predicate in case of any changes (for example when another COBOL dialect uses other statements to modify a variable).

Having the ability to transform COBOL code into XML, we have started to work on two similar declarative approaches to achieve this. In one, we have been experimenting with combining SOUL (Smalltalk Open Unification Language)[5] with this XML representation and then representing the structure of COBOL applications into the logic language SOUL. So far this has allowed for static reasoning only.

The second approach uses a Java-Prolog bridge (in the form of PrologCafe[1]) to enable a Prolog environment to reason about the intermediate XML representation, and even transform it. This has already made it possible to implement a very simple and generic logging aspect.

The above mentioned approaches will end up generating extra COBOL code into the original applications, thereby implementing the proposed aspects. Both approaches allow the exploration of a richer and easier-to-use language to help us express our concerns.

So far we can conclude that a declarative language at a meta level is a powerful medium and that it is certainly suited within this context.

[1] http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/

5 Conclusions

In this position paper we presented our intentions of using dynamic aspects as a technique for semi-automating the process of distilling "lost" business rules out of large pieces of (COBOL) legacy software. These ideas came about within the context of the research project ARRIBA.

We first pointed out the difficulties we came across in this research since we are working on large real-life case studies of COBOL legacy systems. The size and complexity of these information systems and lack of expressiveness regarding modularity cannot be ignored.

We demonstrated how we see our approach being integrated in a simplified but real life problem scenario, frequently encountered in one of the companies involved in our research project. We then advocated the use of a declarative language at the meta-level as some sort of pointcut language to capture specific execution traces. This way we could simplify the process of distilling "lost" business rules out of information systems.

To conclude we would like to point out that, although we have demonstrated our ideas within the context of COBOL legacy applications, we firmly believe that they can be applied to similar cases implemented in other programming languages.

Acknowledgments

Our thanks to Wolfgang De Meuter, Thomas Cleenewerck and Herman Tromp for their ideas and for proofreading the paper.

References

[1] Johan Brichau, Wolfgang De Meuter, Kris De Volder. Jumping Aspects. In ECOOP Workshop on Aspects and Dimensions of Concerns", Cannes, 2000.

[2] Isabel Michiels, Dirk Deridder, Herman Tromp and Andy Zaidman. Identifying Problems in Legacy Software: Preliminary Findings of the ARRIBA Project. In ELISA workshop at ICSM, 2003.

[3] Mo Budlong. Sams Teach Yourself COBOL in 21 Days. Sams Publishing, 1999.

[4] Aberdeen Group. Legacy Applications: From Cost Management to Transformation. Executive White Paper from Aberdeen Group, March 2003. Can be found at http://www.aberdeen.com/2001/research/03038126.asp.

[5] Roel Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. In Proceedings of TOOLS USA '98, 1998.

[6] Mike Murach, Anne Prince, Raul Menendez. Murach's Structured COBOL. Mike Murach & Associates, 2000.

A Generalization and Solution to the Common Ancestor Dilemma Problem in Delegation-Based Object Systems

Eddy Truyen[1] Wouter Joosen[1] Bo Nørregaard Jørgensen[2] Pierre Verbaeten[1]
[1]Department of Computer Science
K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven Belgium
email: [eddy,wouter,pv]@cs.kuleuven.ac.be

[2]Maersk Institute of Production Technology
Southern University of Denmark
DK-5230 Odense M, Denmark
email: bnj@mip.sdu.dk

Abstract

This paper studies the diamond problem in the context of delegation-based object systems. The diamond problem occurs when the same ancestor is inherited multiple times via different inheritance paths. The challenge is that replication and sharing of distinct attributes of the common ancestor must be simultaneously supported. We illustrate the relevance of the diamond problem by showing that it arises not only in multiple inheritance but also in other inheritance techniques, hence the more general term 'the common ancestor dilemma'. More specifically the hybrid approach that integrates object-based inheritance in a class-based model is also affected by the problem. We show that the hybrid approach provides an elegant solution for orthogonal expression of replication and sharing. Attributes that should be shared are modeled as part of the delegating objects, whereas attributes that should not be shared are modeled as part of subobjects of the delegating objects.

1 Introduction

The diamond problem [20], also known as "fork-join" inheritance [18], is a troublesome situation with multiple inheritance which occurs when the same ancestor is inherited multiple times via different inheritance paths, i.e. when two or more ancestors of a class D have a common ancestor A. The question arises whether the attributes (from the common ancestor A) should be inherited in as many versions as there are components deriving from it, or in a single version shared by all components. As argued by [7], both *replication* (i.e. inheriting multiple times) of the meaning of certain attributes and *sharing* (i.e. inheriting once) of the meaning of some other attributes should be simultaneously supported.

We study the diamond problem in the context of delegation-based aspects. Since delegation supports composition at the level of objects, it provides a simple technology for dynamic aspects. Dynamic aspect-orientation is looked upon in this paper as adding or removing aspects to an already running application. A running application consists of a group of collaborating objects that interact with each other by sending messages. An aspect injects components into one or more of these objects and within each object the corresponding component adds new behavior for one

or more operations of that object. Examples of delegation-based aspect technology are JAC [16], Delegation Layers [15], Object Teams [5] and Lasagne [23].

This paper focuses on intra-object composition: we look at the composition of components in one object. Each component stems from a different aspect and the different components are composed by placing them in an incremental modification hierarchy, very similar to a linear mixin-based inheritance hierarchy. This paper addresses the following issues:

1. We discuss the scope of the diamond problem. Mira Mezini referred to the diamond problem in her dissertation [11] as the *common ancestor dilemma* problem. We propose to use this name for the diamond problem because we argue it is a more general problem that does not only arise with multiple inheritance but in every inheritance technique that supports *composition of independently developed components*. Here the sharing versus replication issue will arise in any composition of independently developed components that inherit from a common ancestor, hence the more general name the 'common ancestor dilemma'. Specifically, in aspect-oriented programming when two aspects extend (by means of any available incremental modification relationship) a common aspect, their composition will obviously face the same problem.
2. We highlight the limitations of existing solutions to the common ancestor dilemma problem in the context of *delegation*.
3. We illustrate the strength of hybrid models that integrate *delegation* in a class based programming model. Recent work [6, 2] has elaborated on such an approach. We show that the hybrid approach naturally provides an elegant solution for expressing replication and sharing. As such this solution applies to any delegation based aspect-oriented technology.
4. We document the challenge of separating replicated methods. As argued by [11] replicated attributes must be kept separated from each other in different visibility scopes. Although the hybrid approach effectively resolves ordinary name collisions, it fails to separate replicated methods. Different solutions to this problem exist. We discuss the strengths and weaknesses of each of these solutions

As will be discussed in Section 2 all existing solutions to the original diamond problem incur problems. It is well known that these problems arise because inheritance and visibility control (for the sake of encapsulation) are not orthogonally realized from each other in the design of contemporary programming languages. This lack of orthogonality appears both at the language run-time level and at the programming level. At the language run-time level, the execution environment does not maintain sufficient information to keep the mechanisms apart from each other. At the programming level, wrong programming abstractions are provided so that inheritance and visibility control cannot be expressed in isolation from each other. The existence of this lack of separation of concerns is already well-documented, see for example [13] and [12]. In the line of these thoughts, this paper looks upon the issue to which extent the hybrid approach succeeds in orthogonalizing inheritance and visibility control.

The paper is structured as follows. Section 2 overviews the original diamond problem in the context of multiple inheritance. Section 3 discusses the more general form of the problem and, therefore, proves the relevance of the problem in the context of broad software composition technologies. Section 4 studies the common ancestor problem in the light of delegation and points out why existing solutions needs to be restudied. Section 5 shows how the hybrid approach provides an elegant solution for orthogonal expression of replication and sharing. Section 6 discusses the problem of keeping replicated attributes separated in distinct visibility scopes and discusses the existing solutions to the problem. Section 7 summarizes the paper.

2 The common ancestor dilemma

As shown by [20] the common ancestor dilemma occurs in multiple inheritance when the same ancestor is inherited multiple times via different inheritance paths, i.e. when two or more ancestors of a class D have a common ancestor A. As demonstrated in [7] it is desirable to be able to choose the alternative (replication versus sharing) individually for each attribute. Figure 1 (due to [7, 21]) illustrates this point. When looking at the attributes `name`, `address` and `seniority` in the common ancestor `UniversityEmployee`, there is no doubt that the attributes `name` and `address` should be shared by the `Lecturer` and `AdministrativeStaff` classes. What about `seniority` then? The employee in question has two seniorities, one for each sort of employment. Therefore, the attribute `seniority` should be duplicated in the `Lecturer` and `AdministrativeStaff` classes.

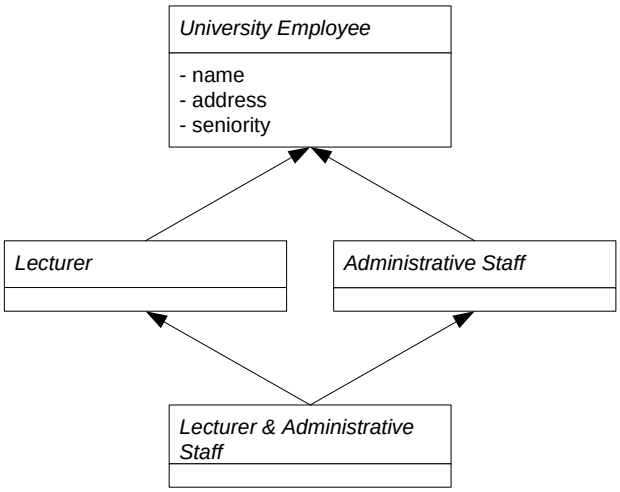

Figure 1. Common ancestor dilemma

Different solutions to the common ancestor problem in multiple inheritance hierarchies have been proposed. None of the approaches is fully satisfactory however (due to [21, 11]):

- graph multiple inheritance: suffers from encapsulation problems and from the undesired duplicate parent operation [20]
- linear multiple inheritance: all replication is simply not allowed to occur.
- tree multiple inheritance: only supports replication [7]

A more general technique is to use renaming [10]. Those attributes that must be replicated are renamed so that there are no name conflicts: those inherited attributes shall be shared that have not been renamed along any of the inheritance paths [18].

Sakkinen [18] refutes all of the above approaches because the common ancestor `UniversityEmployee` gets *effectively split into two*. The problem with this, according to Sakkinen, is that the *integrity* of the independently developed components `Lecturer` and `AdministrativeStaff` *is violated*. Sakkinen defines 'integrity' as the requirement that no property of an object must be changed except by operations that intend and have the right to modify that object.

Sakkinen notes that the "mathematical difference" (i.e. the incremental modification) of a subclass object and a corresponding superclass object is not defined in the conventional inheritance view (i.e. it is embedded directly in the subclass), or in any case it is not an object. To remedy this situation, Sakkinen proposes an inheritance model that is reduced to aggregation, yielding an inheritance model in which the difference will always be an object; but these objects

cannot exist alone, only as part of *complex objects*. He shows that this leads to an inheritance model with much less ambiguous concepts [18].

If we translate the example of Figure 1 into Sakkinen's inheritance model, the `Lecturer&AdministrativeStaff` class is represented by a complex object that aggregates three *subobjects* which are respectively instances of `Lecturer`, `AdministrativeStaff` and `UniversityEmployee`. Returning to the common ancestor problem: in order to accommodate the desired application semantics the `UniversityEmployee` subobject must be split into two: an *S* part that corresponds with the shared attributes and an *R* part that corresponds with the replicated attributes. Figure 2 illustrates this. The integrity of subobjects is thus violated according to Sakinnen. Indeed, if an operation of any of the classes involved ([UniversityEmployee], [Lecturer] or [AdministrativeStaff]) updates shared attributes, based on the value of a replicated attribute, then invariants or assumptions that have been made about the children ([Lecturer] and [AdministrativeStaff]) may break. In other words all operations must be checked in order to identify and resolve such problems; thus the advantage of inheritance is lost.

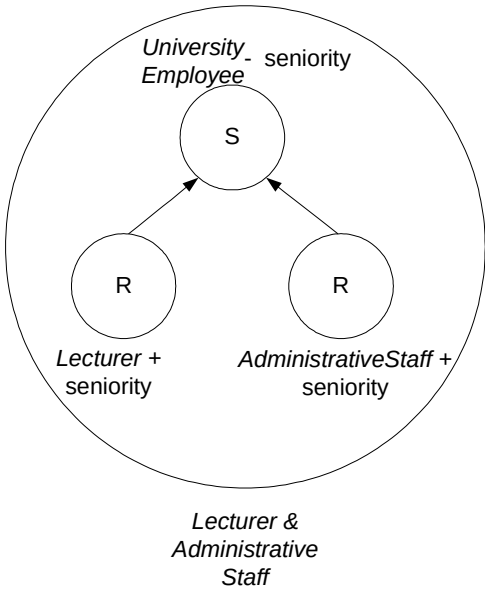

Figure 2. Splitting the common ancestor

In order to circumvent integrity violation Sakkinen argues that the application designer must explicitly divide the common ancestor class into two classes in the first place: `Person` (containing `name` and `age`), and `Only-UniversityEmployee`(=`UniversityEmployee`-`Person`, containing the attribute `seniority`). `Person` is then a shared parent of `Only-UniversityEmployee`. `Lecturer` and `AdministrativeStaff` would inherit without sharing from `Only-UniversityEmployee` [18]. Figure 3 illustrates this. Although this is a clean approach, it puts a burden on the application designer because it implies that the common ancestor dilemma must be anticipated at class design time.

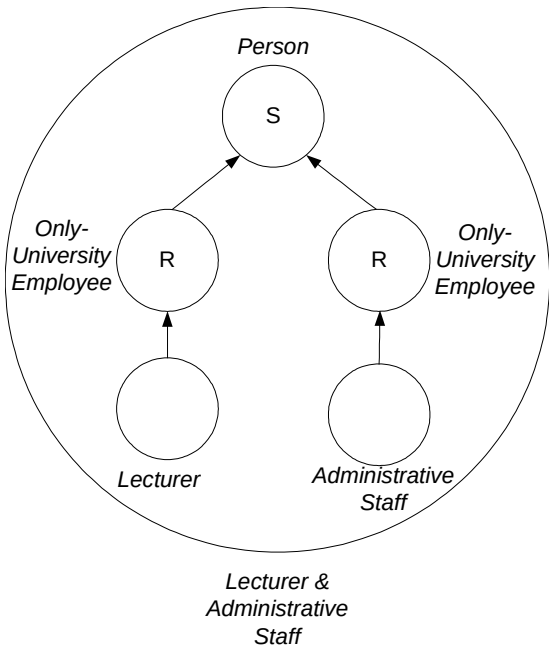

Figure 3. Splitting the common ancestor during class design

3 Generalization of the problem

The existing literature we have studied [7, 18, 21, 11, 19] indicates that the diamond problem is very difficult to solve for state attributes. As such, it is tempting to ignore multiple inheritance: given the fact that the common ancestor problem seemingly only appears with multiple inheritance, the problem could simply be side-stepped by discarding multiple inheritance as a useful software composition tool because it inherently suffers from implementation problems [3,22] and conceptual problems [19] anyway. This would however be the wrong thing to do as argued by the following points.

We observe that the common ancestor dilemma appears with *any inheritance approach that supports composition of independently developed components*. Indeed, if two independently developed components, *that inherit from a common ancestor*, are composed (by either multiple inheritance, mixin-based inheritance [1], or any other eligible inheritance technique), the problem arises. For example, suppose it was possible to specify explicit inheritance relationships between mixin-classes, then mixin-based inheritance would also have to deal with the problem. Figure 4 illustrates this. Consequently the original diamond problem, identified in multiple inheritance, is an instance of the more general 'common ancestor dilemma' problem.

Another, but less important point is the usefulness of *repeated inheritance* as a compositional tool. As indicated by [10], inheritance of the same ancestor via different paths is a generalization of repeated inheritance, where the same parent class is directly inherited multiple times. To better distinguish the two cases, the former case is also called *indirect repeated inheritance*, while the latter *direct repeated inheritance*. Direct repeated inheritance is clearly useful as a compositional tool. For example, one class can be explicitly inherited twice to implement two similar, but distinct features at the object level (for example a student that is also an employee at our university has two `MemberID` attributes; Both of the attributes may be instantiated from the same class, but their respective values are necessarily different [21]). Consequently we cannot ignore the common ancestor dilemma problem because otherwise the usefulness of direct repeated inheritance would be wasted.

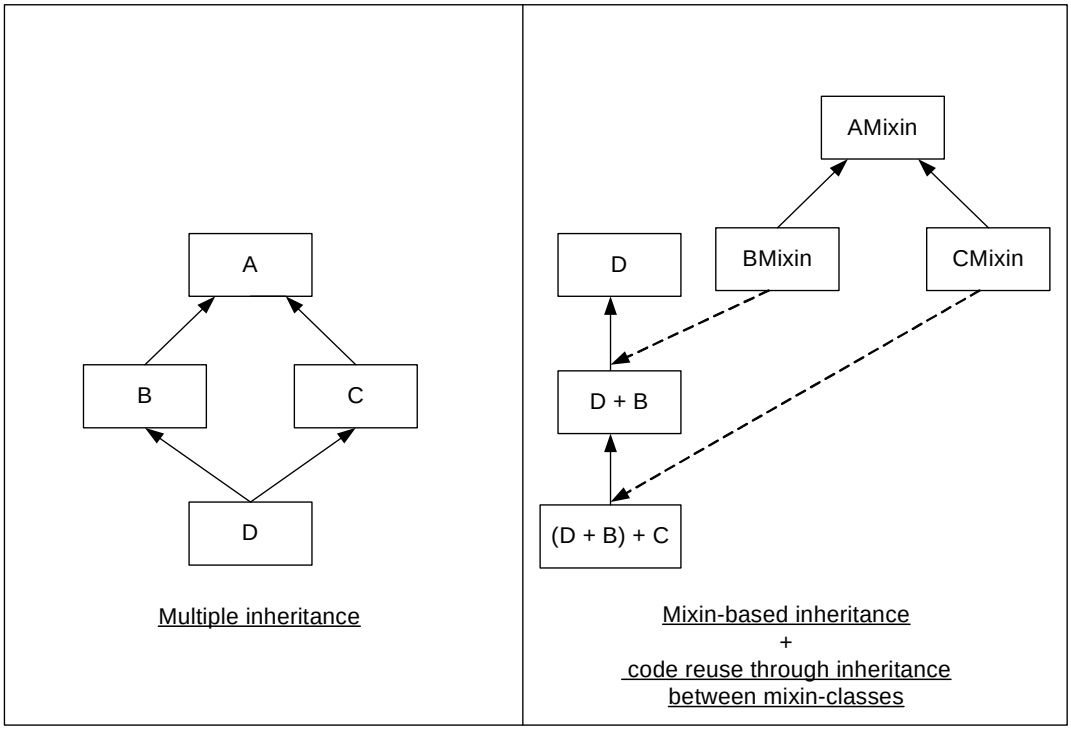

Figure 4. Diamond problem and common ancestor dilemma problem

The above two points therefore prove the relevance of the common ancestor dilemma problem in the context of a broad range of software composition technologies. In fact, the problem also arises in delegation-based object systems. The next section discusses this in further detail and explains why solutions to the common ancestor dilemma problem must be revisited in the light of delegation and, therefore, dynamic aspects.

4 The common ancestor dilemma in the context of delegation

This section studies how the common ancestor dilemma arises in programming languages that support delegation. We will show that the problem of integrity violation, as pointed out by Sakkinen, is irrelevant in the light of delegation, because it is superseded by another problem. Hence, we investigate what is a good solution to the common ancestor dilemma problem in the light of delegation. Before we proceed with the discussion, we first introduce delegation and explain the kinds of delegation that are relevant in the context of aspect-orientation.

4.1 Delegation

Delegation was originally introduced by Lieberman [8] in the framework of a classless prototype-based language. Delegation allows the behavior of an object to be defined in terms of the behavior of another object. An object, called the *child*, may have modifiable references to other objects, called its *parents*. A message for which the receiving object has no matching method are automatically forwarded to one of its parents, that responds on behalf of the receiver. When a suitable method is found in the parent object (the *method holder*) it is executed after binding its implicit *self* parameter. This parameter refers to the message receiver on whose behalf the method is executed. Automatic forwarding with binding of self to the message receiver is called *delegation*. Automatic forwarding with binding of self to the method holder is called *consultation* [6].

There are two forms of delegation. Static and dynamic delegation. In dynamic delegation the parent of an object can dynamically change. Here the parent of an object is typically stored in some specially identified instance variable. This instance variable can be consulted and also modified, thus changing an object's parent. With static delegation, the parent object must be assigned when the object is created and cannot be reassigned during the object's lifetime. Furthermore, with multiple delegation a child object can have multiple parent objects, whereas with single delegation a child object can only have one parent object.

Delegation can also be interpreted as an incremental modification mechanisms and, therefore, delegation has also been called *object-based inheritance* [21]. Child and parent respectively correspond with inheriting client and ancestor.

4.1.1 Hybrid approaches

Delegation has recently regained a lot of interest as part of a *hybrid approach* that integrates delegation in a class-based model. The hybrid approach has regained interest because of its powerful, yet type safe use in the context of class-based programming languages, demonstrated by Lava [6], and support for type transparency, demonstrated by the Generic Wrappers approach [2].

To illustrate the concepts in the remainder of this paper as concretely as possible, we will use a concrete programming model that integrates delegation with the class-based programming model. With this end in view we take the programming model of the Generic Wrappers approach [2] which we will shortly overview here.

Parent and child objects are declared as normal classes. The parent object and each of its child objects may be associated to a separate (class-based) inheritance hierarchy. For example:

```
public class DeclaredParent {
   public void b();
}

public class Parent extends DeclaredParent {
   public void b() {
      ...
      super.b();
      ...
   }

  public void foo() { ... }
}
```

Child objects are classes that are declared to wrap instances of a given reference type (class, interface) or of a subtype thereof. The wrapped instance is called the *wrappee*. Like an `extends` clause to specify a superclass, a `wraps` clause is used to state the static wrappee type. This also declares the wrapper class to be a subtype of the static wrappee type. For example the declaration

```
public class Child wraps DeclaredParent {}
```

states that each instance of the class `Child` wraps an instance of a class DeclaredParent or of any subtype thereof. The declaration makes Child a subtype of DeclaredParent. Thus, instances of Child can be assigned to variables of type DeclaredParent and Child has all public members of DeclaredParent.

To assure that this subtyping relationship always holds (and thereby that forwarding of calls never fails) instances of Child must always wrap an instance of DeclaredParent or a subtype thereof - already during the execution of constructors. Hence the wrappee must be passed as a special argument (in the syntax of [2] by `< >`) to class instance creation expressions

```
Parent p = new Parent(...);
DeclaredParent c = new Child(...)<p>;
```

Figure 5 shows a class diagram[1] that graphically represents the program listed above where delegation links correspond with the `wraps` clauses and inheritance links correspond with the `extends` clause.

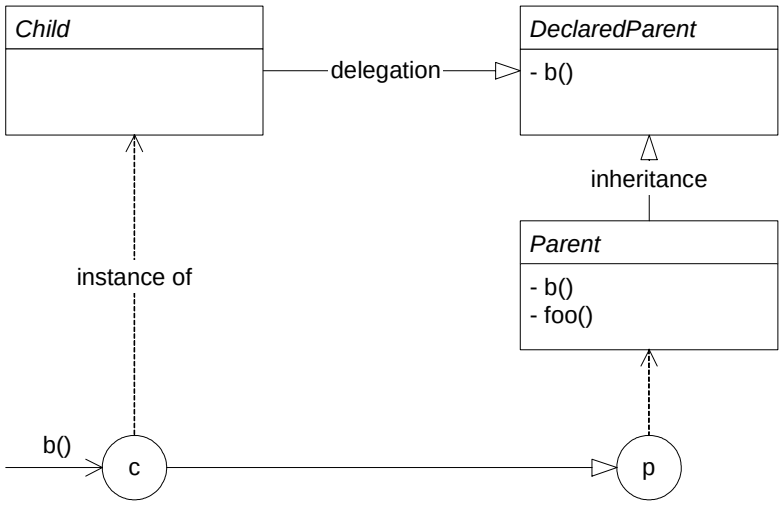

Figure 5. Delegation in a class-based programming model

Delegation is illustrated by the fact that method `b()`, declared in `DeclaredParent`, can be called on the `Child` object. This is illustrated in the following program fragment, which is based on the program listed above. Furthermore, since delegation enables late binding of self, the `b()` method of `Parent` is actually executed:

```
c.b();
```

A particularity of the hybrid approach is its support for type transparency. This means that child objects are not only of the static, but also of the actual wrappee type. For example, a `Child` object wrapping a `Parent` object is also of the latter type and not just of type `DeclaredParent`. Hence, such an aggregate can be assigned to a variable of type `Parent` and the latter's methods can be called on it. In the following program fragment, which is based on the definition of `Child` above, the type test returns true and the cast succeeds:

```
if (c instanceof Parent) {
  ((Parent)c).foo();
}
```

[1] The graphical notation of the figure is due to [6].

4.1.2 Delegation and dynamic aspects

The hybrid approach has especially gained a lot of interest as a simple technique for dynamic composition of aspects. An aspect could be modeled as a set of child classes. An aspect can then be injected in an already running application by placing instances of these child classes around different application objects that play the role of parent object. Furthermore, there already exist various approaches that lift delegation to real dynamic object modification. For instance, approaches such as JAC [16], Delegation Layers [15], Object Teams [5] and Lasagne [23] free the programmer from having to manually interpose a child object around a parent object through explicit object reference switching.

Given the subject of this paper, we only focus on intra-object composition: the composition of multiple child objects around a single parent object. Each child object stems from a different aspect and the different child objects are composed by placing them in a linear incremental modification hierarchy (also known as conjunctive wrapping), very similar to mixin-based inheritance hierarchies [1]. Of course this statement indicates that the topic of this paper is a general language design issue that has only marginally to do with aspects. However, when looking through an aspect-oriented lens, the relevant design space of delegation-based systems becomes considerably smaller. First, we do not regard delegation in the arena of conceptual modeling but as a tool for composing independently developed components. In other words we study delegation as a composition operator, not as a specialization or an "is-a" relationship. Second, this paper takes single delegation as the basic intra-object composition operator because every child object is meant to extend only one parent object.

4.2 Revisiting the common ancestor dilemma problem

Single delegation also has to deal with the common ancestor dilemma. When a child and a parent object are composed by means of delegation, the common ancestor dilemma arises in one of the following two cases as illustrated in Figure 6:

- **(a)** inheritance from a common declared superclass
- **(b)** delegation to a common declared super type

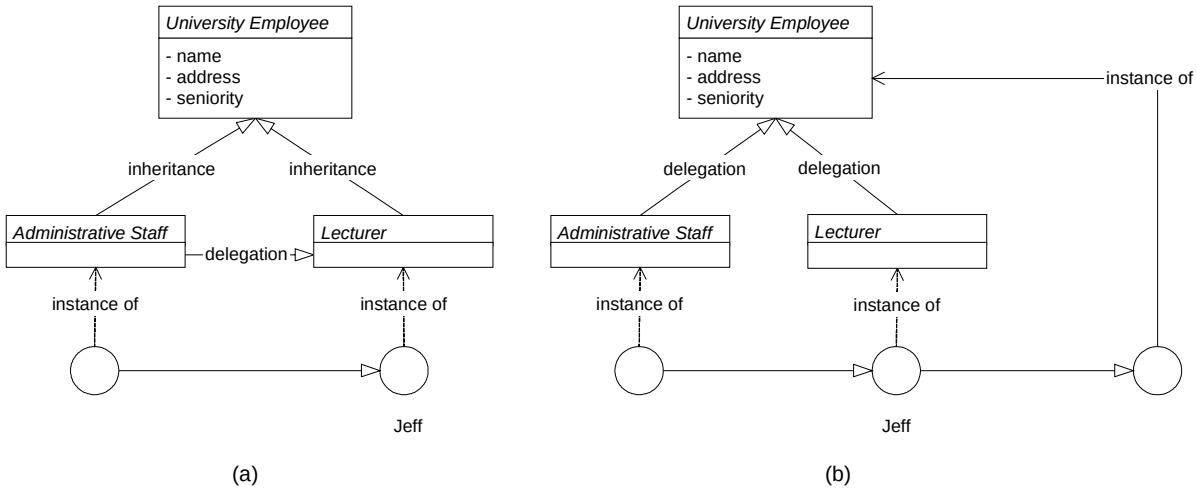

Figure 6. Delegation and the common ancestor dilemma

Since we studied the common ancestor dilemma in the context of Sakkinen's inheritance model [18] (see Section 1), we first have to map Sakkinen's model to delegation. This is quite easy to do because Sakkinen reduces inheritance to aggregation: the notion of complex object

corresponds with the dynamically bound common self across the web of parent and its child objects. A subobject in that complex object corresponds with the parent or one of its child objects.

We will now explain how the problem focus is shifted in the light of delegation. Suppose Jeff who is a `Lecturer` is asked to handle some administrative task as well. To accommodate this situation in real-time, a reconfiguration must take place at run-time: the complex object representing Jeff must be dynamically extended with an `AdministrativeStaff` subobject. As argued in Section 1 it is desirable to be able to choose between replication and sharing individually for each attribute of the common ancestor `UniversityEmployee`. In case (a), however, replication is the obligatory default for all the common ancestor's attributes, while sharing is in case (b). In case (a) Jeff would have duplicate name and address attributes which is undesirable from a conceptual modelling standpoint (the name and home address of a person are conceptually unique) and from a state consistency standpoint (clients accessing different subobjects must observe and modify (through getters and setters methods) the accidentally duplicated attributes in a mutually consistent fashion) In case (b) Jeff would have the same seniority for both sorts of employment which is obviously undesired from a conceptual modelling standpoint. So also here, the `UniversityEmployee` subobject of Jeff needs to be split into two.

However, splitting of the common ancestor simply *cannot* be performed because the composition operator provided by delegation operates at run-time, at the level of *operational* subobjects. Technically speaking, splitting a subobject after creation (i.e. at run time) is not feasible.

As such, when already running application objects (complex objects in the terminology of Sakinnen) are to be modified over time with subsequent new components, the common ancestor dilemma may strike at any point of time when any pair of two components share a common ancestor. If the dilemma occurs, however, then the common ancestor has been instantiated already. The ancestor therefore cannot be split anymore.

5 Towards an elegant solution

We consider three alternative approaches to address the aforementioned problem.

The first approach would address a solution that disallows specifying any explicit inheritance relationship. The idea is that by disallowing specification of explicit inheritance relationships the common ancestor dilemma will not occur in the first place and as such no solution would be required. Pure mixin-based inheritance (where explicit inheritance relationships between mixin classes is not allowed) is an example. However this is a fake solution because, as argued in Section 3, direct repeated inheritance faces the same issues of the common ancestor dilemma. As such the problem still needs to be solved in order to gain the expressive power of direct repeated inheritance.

A second approach would be to advance the technological state-of-the-art in virtual machine support to allow splitting subobjects while the complex object is running. We think working towards this goal is neither realistic nor a good idea. First, there are the problems related to integrity violation as mentioned by Sakinnen. Secondly, although there exists work about run-time support for changing classes [9] or adding aspectual behavior during execution [17], splitting classes at run-time seems extremely difficult to do without incurring a lot of other problems.

The third and only option left for dealing with the common ancestor dilemma in the light of delegation is the original solution from [18]: namely to side-step the splitting problem by explicitly dividing the common ancestor into two classes *S* and *R* during software design. The class *S* contains attributes to be shared and the class *R* is to be replicated (see Figure 3). What is obviously needed is the expressive power that enables the modelling of such an ancestor structure.

This is where the power of the hybrid approach comes into play. It provides a natural solution for respectively expressing sharing and replication without interfering with each other.

Sharing is realized by means of delegating to the *S* subobject, while replication is realized by means of inheriting the *R* class. The code below (based on the programming model introduced in Section 4.1.1) illustrates how a structure similar to Figure 3 can be easily implemented: class `UniversityEmployee` is effectively split in two classes during class design: class `Person` encodes the *S* part and `Only-UniversityEmployee` encodes the *R* part. Since university employees are persons, class `Only-UniversityEmployee` obviously needs to extend the Person class. Since the `Person` class represents the *S* part, this incremental modification should be expressed by means of delegation.

```
public class Person {
   private Name name;
   private Address address;

   public String getName() {...}
   public Address getAddress() {...}

}

public class Only-UniversityEmployee wraps Person{
   private Seniority seniority;

   public Seniority getSeniority() {...}

}
```

The classes, representing different forms of employment, however, need to be defined as an incremental modification by means of class-based inheritance because `Only-UniversityEmployee` needs to be replicated.

```
public class Lecturer extends Only-UniversityEmployee {
   String title;

   public Lecturer(String title) {
     this.title = title
   }

   public String getName() {
      return title + super.getName();
   }

  public void foo() {...}
       self.getSeniority();
  }
}

public class AdministrativeStaff extends Only-UniversityEmployee
{
   String jobtitle;

   public void bar() {...}
       self.getSeniority();
  }

  public String getName() {
    return super.getName() + ", " + jobtitle;
}
```

Finally, the scenario in Section 4.2 that `Lecturer` Jeff is suddenly employed as `AdministrativeStaff` can easily be accommodated by wrapping the jeff object and reassigning the result to the `jeff` variable.

```
//main
Only-UniversityEmployee jeff = new Lecturer("Prof. dr".)<new
Person(...)>;

...

jeff = new AdministrativeStaff(...)<jeff>;

((Lecturer)jeff)foo();
```

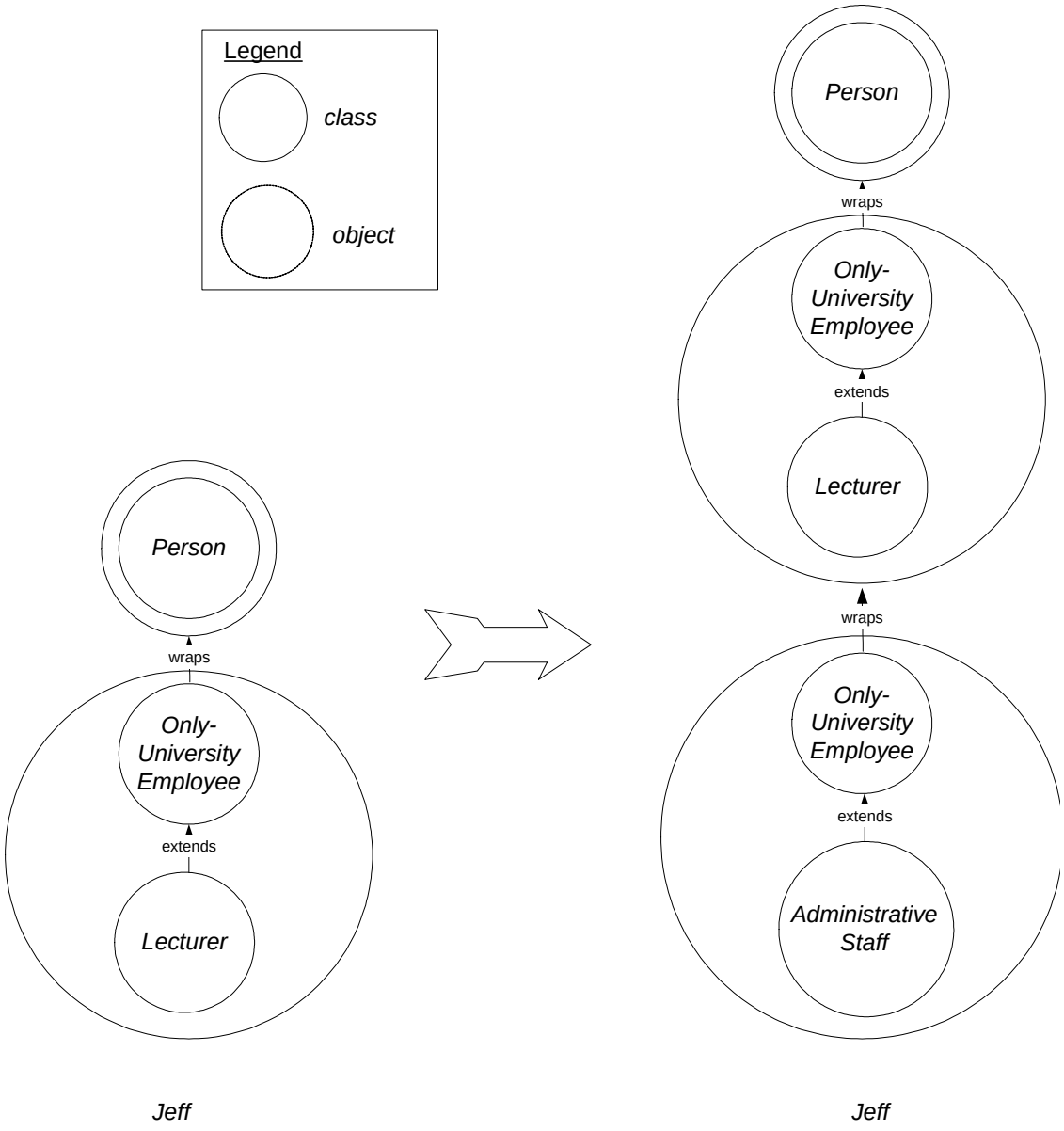

Figure 7. Jeff becomes administrative staff

This is further illustrated in Figure 7. Before wrapping it with `AdministrativeStaff`, complex object `jeff` consists of an object hierarchy of which the `Lecturer` subobject encloses an `Only-UniversityEmployee` subobject. After wrapping complex object `jeff`, both `Lecturer` and `AdministrativeStaff` subobjects contain a duplicate of the `Only-UniversityEmployee` class.

6 Keeping replicated attributes separated

Replicated attributes must be kept separated from each other because they are considered to belong to two different subobjects of a single complex object. Coping with this problem equals at first sight to the problem of coping with name collisions in general that occur when two independently developed components accidentally use the same name for different attributes; Attributes involved in such ordinary name collisions are often called *homonymous attributes* [11].

Although the hybrid approach effectively deals with ordinary name collisions, it fails to separate replicated methods as will be explained. This section subsequently discusses various existing solutions to this particular problem.

6.1 The problem

In the hybrid approach replicated state variables can easily and effectively be kept separated from each other if they are declared as non-public attributes. Dealing with replicated methods is more difficult as will be explained below.

To deal with homonymous methods in the context of delegation, Günter Kniesel proposes the following adapted rule for method overriding [6]:

> For a message *recv.n*(*args*) (i.e. `self.getSeniority()`) a method with signature σ (i.e. `getSeniority()`) from type *T* (i.e. `AdministrativeStaff` overrides the matching method from the static type of *recv*, T_{stat}(i.e. `Lecturer`) if there is some common declared supertype of *T* and T_{stat} (i.e. `Only-UniversityEmployee`) that contains σ.

The adapted rule for method overriding essentially boils down to the existence of *a common declared supertype*. This exact feature, however, renders the adapted rule completely useless for keeping replicated methods separated from each other. This is because replicated methods are declared by a common ancestor class and, therefore, the adapted rule would incorrectly enable overriding between replicated methods. The adapted method rule does work for homonymous methods because these methods stem from different unrelated ancestors.

Let us apply the adapted rule to the running example to illustrate our point. Consider in Figure 7, the complex object `jeff`, that consists of a `Person` subobject, a `Lecturer` subobject with an enclosed `Only-UniversityEmployee` subobject, and an `AdministrativeStaff` subobject with a second enclosed `Only-UniversityEmployee` subobject. Here overriding between the `Lecturer`-specific and `AdministrativeStaff`-specific methods of the `getSeniority()` operation is enabled according to the adapted rule. This is because there exists a common declared supertype of `Lecturer` and `AdministrativeStaff` (i.e. `Only-UniversityEmployee`) that declares `getSeniority()`. As a result, the self call `to getSeniority()` depicted in the above example from within the `Lecturer` subobject will be incorrectly redirected to the `AdministrativeStaff` object. As such it is clear that the adapted rule for method overriding incorrectly enables overriding between replicated methods. Note that the adapted rule also breaks in the case of direct repeated inheritance.

6.2 Existing solutions

This section discusses various solutions to the problem of keeping replicated methods separated in the hybrid approach. Since our solution of expressing replication and sharing is based on a redesign, it is worth to consider other redesign options for maintaining replicated methods in mutually invisible scopes. We also investigate solutions based on additional machinery in programming language. Basically, a good solution makes a good trade-off between providing additional machinery that the average programmer can understand, and creating software designs that are easy to maintain and evolve.

6.2.1 Disjunctive wrapping

An astute reader might notice that replicated methods can easily be kept separated from each other by disjunctively wrapping the `AdministrativeStaff` and `Lecturer` objects (instead of conjunctively wrapping them). One could indeed have one instance of `Lecturer` and one instance of `AdministrativeStaff`, each delegating to a `Person` object, but not to each other. By letting the client have different references “jeffAsLecturer” and “jeffAsAdministrator” to the two delegating objects, one could already achieve the desired separation between replicated methods. This of course also works for self calls because self is dynamically bound to the original message receiver.

This solution is very elegant from a conceptual modeling point of view and is also in line with a frequently occurring situation in role-based design. In the latter it frequently occurs that an object plays different roles in different contexts and within each context the object plays never more than one role. The role an object plays depends thus on the context in which the object is currently being used.

From a compositional point of view, however, disjunctive wrapping does not support combination of methods that override a method of the shared part *S*. In the running example both `Lecturer` and `AdministrativeStaff` override the `getName()` operation of `Person`. In a disjunctive wrapping style it is not possible to invoke the full-combined behavior of both overriding methods.

6.2.2 Replacing class-based inheritance with aggregation

Another solution is to model the *R* part of the common ancestor as a “real aggregated” subobject of the delegating objects. Thus instead of inheriting the *R* part one aggregates the *R* part. Suppose in the running example `Lecturer` and `AdministrativeStaff` would directly delegate to `Person`, whereas the `Only-UniversityEmployee` class is completely independent of the `Person` hierarchy. Then replication could simply be expressed by having the delegating objects aggregate a different `Only-UniversityEmployee` object. Having moved the "seniority feature" to different aggregated subobjects, we obviously do not get any overriding of `getSeniority()` methods any more.

This approach solves the issue of separating replicated methods quiet nicely for self calls. For non-self calls, however, there is the problem of the necessary plumbing that must be manually programmed in order to allow clients access to the appropriate replicated subobject. This also implies that the client has to manually navigate through the delegation hierarchy to find the appropriate subobject he is currently interested in.

6.2.3 Multiple delegation from a proxy object

Letting a surrogate / proxy object multiply delegate to the `Lecturer` and to the `AdministrativeStaff` object (which themselves share the identical `Person` parent) expresses exactly the desired sharing and replication semantics and lets it simply be understood from the shape of the delegation hierarchy as depicted in Figure 3. The desired separation

between replicated methods can be easily achieved in multiple delegation by using the “sender path tiebreaker rule", used in the design of the Self programming language [25], or by renaming of selectors [10], used in the design of Lava [24]. Moreover, these solutions nicely complement the disjunctive wrapping style. Combination of methods that override a shared method can be accommodated by explicit local redefinitions at the proxy.

The disadvantage of the approach is that the creation of the proxy class, renaming and explicit local redefinitions puts an extra burden on the programmer. Furthermore, the approach is not scalable enough to cope with the situation that multiple common ancestor dilemmas must be resolved within the same complex object.

6.2.4 Scope identifiers

The Rondo object model [12, 11] effectively supports separating replicated methods by means of a mechanism based on so called *scope identifiers*. This mechanism is uniform in the sense that it resolves both kinds of conflicts (replicated methods as homonymous methods) in identical the same way. Although the Rondo model has not been developed in the context of delegation-based systems, a mapping to delegation is straightforward. Scope identifiers are constructed as follows: each child object is marked with a unique label and the scope identifier of a method simply concatenates the labels of child object that are in the visibility scope of that method.

Although the Rondo model is elegant from a language run-time engineering point of view, the mechanism of scope identifiers does not provide the right abstraction for dealing with name collisions that occur when non-self calls are sent from message-passing clients. This is because the labels of child objects that are used to construct scope identifiers are implicitly generated by the internal structures of the Rondo engine and therefore do not have a meaning in the domain of message-passing clients. As such it seems that although the mechanism of scope identifiers sufficiently applies separation of concerns at the language design space to effectively control distinct visibility scopes for replicated methods, the mechanism does not have a meaning to message-passing clients.

The other solutions discussed above do not suffer from this problem because they model visibility scopes in the domain of the application and, therefore, have a clear meaning in the domain of message-passing clients. We believe that a programming language whose execution environment is based on the Rondo model and whose programming model provides sufficient expressive power to model scope identifiers in the domain of the application is a very powerful solution. Future work in this context is to study to which extent the notion of dependent types [4, 14, 15] is feasible to serve this purpose. The notion of dependent types implies that an object can aggregate one or more inner classes that are virtual, meaning that the type of these inner classes is dependent on the identity and type of the aggregating outer object.

7 Conclusion

This paper has focused on dynamic intra-object composition. We look at the composition of aspects in one object. Each component stems from a different aspect and the different components are composed by placing them in an incremental modification hierarchy, very similar to a linear mixin-based inheritance hierarchy.

We have discussed the scope of the common ancestor dilemma problem from this perspective. Specifically, in aspect-oriented programming when two aspects extend (by means of any available incremental modification relationship) a common aspect, their composition obviously faces a similar problem. We have highlighted the limitations of existing solutions to the common ancestor dilemma problem in the light of dynamic aspects. We have illustrated the strength of hybrid models that integrate delegation in a class based programming model. We have shown that the hybrid approach naturally provides an elegant solution for expressing replication and sharing. As such this solution applies to any delegation-based aspect-oriented technology. We

have documented the challenge of separating replicated methods and we have discussed the existing solutions to this problem.

Acknowledgments

Many of the ideas in this paper were inspired by the discussion of the common ancestor dilemma in the dissertations of Mira Mezini [11] and Patrick Steyaert[21]. The structure of the paper was helped along by discussions with Nico Janssens. We would like to thank the anonymous reviewers for their very informative and helpful comments and for pointing out some of the existing solutions to separating replicated methods.

References

[1] Gilad Bracha and William Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*, pages 303–311, 1990.

[2] Martin Büchi and Wolfgang Weck. Generic Wrappers. In Elisa Bertino, editor, *ECOOP 2000, 14th European Conference on Object-Oriented Programming*, volume 1850 of *LNCS*, pages 201–225. Springer-Verlag, 2000.

[3] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA'89 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, pages 211–214. ACM Press, 1989.

[4] Erik Ernst. Family polymorphism. In *ECOOP 2001—Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer-Verlag, June 2001.

[5] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer, October 2002.

[6] Günther Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP '99—Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 1999.

[7] Jørgen Lindskov Knudsen. Name Collision in Multiple Classification Hierarchies. In S. Gjessing and K. Nygaard, editors, *Proceedings of the ECOOP '88 European Conference on Object-oriented Programming*, LNCS 322, pages 93–109. Springer Verlag, August 1988.

[8] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA'86*, pages 214–223. ACM SIGPLAN Notices 21(11), 1986.

[9] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *ECOOP 2000 – Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer-Verlag, June 2000.

[10] B. Meyer and N. Wilson. Eiffel: The Reference. Technical Report TR-EI-41/ER, ISE, 1995.

[11] M. Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publishers, 1998.

[12] Mira Mezini. Dynamic object evolution without name collisions. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 190–219. Springer, 1997.

[13] Oscar Nierstrasz and Dennis Tsichritzis. *Object-Oriented Software Composition*. Prentice-Hall, 1995.

[14] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003—Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer-Verlag, July 2003.

[15] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP 2002—Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110. Springer-Verlag, 2002.

[16] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer, September 2001.

[17] A. Popovici, G. Alonso, and T. Gross. Just in time aspects: Efficient dynamic weaving for java. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*. ACM press, March 2003.

[18] Markku Sakkinen. Disciplined Inheritance. In *Proceedings of the ECOOP '89 European Conference on Object-oriented Programming*, pages 39–56, Nottingham, July 1989. Cambridge University Press.

[19] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behavior. In *ECOOP 2003 - Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274, July 2003.

[20] Alan Snyder. Inheritance and the development of encapsulated software components. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, Series in Computer Systems, pages 165–188. The MIT Press, 1987.

[21] P. Steyaert. *Open Design of Object-Oriented Languages. A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, Belgium, 1994.

[22] Peter Sweeney and Joseph Gil. Space-and time-efficient memory layout for multiple inheritance. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10) of *ACM Sigplan Notices*, pages 256–275, N. Y., November 1–5 1999. ACM Press.

[23] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Nørregaard Jørgensen. Dynamic and selective combination of extensions in Component-Based applications. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*, pages 233–242. IEEE Computer Society, May12–19 2001.

[24] Kniesel, *Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems*, Technical Report IAI-TR-94-3, Computer Science Department III, University of Bonn, Oct. 1994 (revised April 1995).

[25] C. Chambers, D. Ungar, B.-W. Chang and U. Hölzle, "Parents and shared parts of Objects: Inheritance and Encapsulation in SELF," *Lisp and Symbolic Computation: An International Journal*, vol. 4, no. 3, 1991, pp. 21-36.

Optimizing JAsCo dynamic AOP through HotSwap and Jutta.

Wim Vanderperren
Davy Suvée
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 62
wvdperre@vub.ac.be

Abstract

The main drawback of all dynamic AOP technologies available today is the rather high performance overhead in comparison to static weaving approaches. In this paper, we propose an approach to improve the performance of both the interception mechanism and the aspect interpreter of a dynamic AOP system. The interception of the base application is optimized by employing the Java HotSwap technology in such a way that only those joinpoints where aspects are applied upon are trapped. When new aspects are added, all corresponding joinpoints are hot-swapped for a trapped version. Likewise, when aspects are removed, the corresponding traps are removed, if no other aspect is applicable at the given trap. In order to improve the aspect interpreter, we propose the Jutta system that allows generating and caching a highly optimized code fragment for each joinpoint. This code fragment contains the combined aspectual behavior for the joinpoint at hand. We integrate HotSwap and Jutta in the JAsCo dynamic AOP system and perform extensive benchmarks to evaluate the performance gain of this approach. In addition, the enhanced JAsCo performance is compared to a selection of current state-of-the-art dynamic AOP approaches. These benchmarks indicate that JAsCo, enhanced with HotSwap and Jutta, is able to improve on the current state-of-the-art performance-wise.

1 Introduction

AspectJ is undoubtedly one of the most well-known and mature aspect-oriented approaches available today [1]. AspectJ employs static weaving in order to combine the base program and the aspects. As such, aspects cannot be added or removed at run-time; the application needs to be stopped, compiled and restarted in order to change the aspectual behavior. Aspects however often represent concerns that have to be enabled, altered and disabled quite frequently. Typical examples of such crosscutting concerns are debugging concerns such as logging [12] and contract verification [22], security concerns [23] such as confidentiality and access control, management concerns [24] such as accounting and billing, and business rules [5,15] that describe business-specific logic.

During the last years, a wealth of approaches have been proposed to increase the dynamicity of aspect-oriented programming. Examples include PROSE1&2 [18,17], WOOL [19], JAC [16], EAOP [6], OIF [7], AspectWerkz [3], JBoss/AOP [8], HandiWrap [2], AspectS [9], Caeser [14] and JAsCo [21]. The main drawback of all these approaches is the rather high performance overhead required for applying aspects dynamically in comparison to statically weaved languages like AspectJ. This overhead stems from 1) the interception system employed to interfere with the

regular application execution and 2) the aspect interpreter that evaluates which aspects are available at a certain joinpoint and which executes the appropriate advices. In this paper, we investigate how these two mechanisms can be optimized. In order to improve the interception system, we propose to employ the novel Java HotSwap technology that allows replacing the byte code of a class at run-time. In order to improve the second phase, namely the aspect interpretation part, we propose a generic dynamic AOP optimizer, named Jutta. Jutta enables to generate highly optimized code fragments that contain the combined aspectual behavior for each joinpoint. As a proof of concept, we integrate Jutta and HotSwap in the JAsCo dynamic aspect-oriented programming language.

The next section introduces the JAsCo aspect-oriented approach and elucidates the dynamic AOP features offered by this approach. Section 3 presents the Jutta approach and section 4 illustrates JAsCo HotSwap. In section 5, a detailed performance evaluation is performed that compares the enhanced JAsCo implementation with the original JAsCo implementation and a selection of current state-of-the-art dynamic AOP approaches. Finally, section 6 discusses related work and section 7 states our conclusions.

2 Introduction to JAsCo

JAsCo is a dynamic AOP approach originally aimed at combining ideas of aspect-oriented and component-based software engineering. The next sections shortly present the JAsCo approach and discuss the main dynamic features of JAsCo. For more detailed information about JAsCo, the interested reader is referred to [21].

2.1 JAsCo language

JAsCo is mainly based upon two existing approaches: AspectJ and Aspectual Components [13]. The JAsCo language is an aspect-oriented extension of Java that stays as close as possible to the original Java syntax and concepts and introduces two additional entities: *aspect beans* and *connectors*.

```
1  class CachingManager {
2    Cache cache = new Cache();
3    void setRecyclingRate(int sec) {
4      cache.recylingRate(sec);
5    }
6
7    hook CacheControl {
8     CacheControl(method(..args)) {
9       execute(method);
10     }
11
12     replace() {
13       if(cache.isCached(method,args) {
14         return cache.getCached(method,args);
15       }
16       else {
17         Object result = method(method,args);
18         cache.cache(method,args,result);
19         return result;
20       }
21     }
22   }
23 }
```

Figure 1. A JAsCo aspect bean for caching.

An aspect bean is an extended version of a regular Java bean and is specified independent of concrete component types and APIs, making it highly reusable. An aspect bean contains one or more logically related hooks that describe the crosscutting behavior itself. Hooks are able to define three types of advice, namely before, replace and after, which are equivalent to the before, around and after advices known from AspectJ. Figure 1 illustrates an aspect bean that captures a caching concern. The crosscutting behavior, namely intercepting the invocation and returning a cached result instead, is captured in the *CacheControl* hook. Also notice the special constructor of a hook, which specifies a kind of abstract pointcut (line 8 till 10).

A connector on the other hand, is used for deploying one or more aspect beans within a concrete component context As such, a connector allows to explicitly instantiate and initialize hooks. In addition, connectors are able to specify explicit precedence and combination strategies in order to manage the cooperation among several aspects that are applicable onto the same joinpoint. Figure 2 shows a connector that instantiates the *CacheControl* hook of Figure 1 onto the *getHotels* method of a *BookHotel* component.

```
1  static connector CachingConnector {
2
3    CachingManager.CacheControl ca =
4      new CachingManager.CacheControl (
5        List BookHotel.getHotels(String)
6    );
7
8    ca.setRecylingRate(60);
9    ca.replace();
10 }
```

Figure 2. A JAsCo connector deploying the caching aspect bean of Figure 1.

2.2 JAsCo technology

In order to implement the JAsCo language, we propose a new component model where traps that enable aspect interaction are already built-in. Ideally, new components are shipped employing this new component model. This way, attaching and removing aspects to components implemented in the new component model does not require any adaptation whatsoever to the target beans. Of course, expecting all components to be developed using this new component model is rather utopian. Therefore, it is also possible to automatically transform a regular Java bean into a JAsCo bean by employing a preprocessor that inserts the traps using byte-code adaptations.

Each trap refers to the JAsCo run-time infrastructure that manages the registered connectors and aspect beans. Figure 3 illustrates the run-time infrastructure schematically. The central connector registry serves as the main addressing point for all JAsCo entities and contains a registry of connectors and instantiated hooks. The connector registry is notified when a trap is reached or when a connector is loaded. As such, the database of registered connectors and hooks is updated dynamically. The left-hand side of Figure 3 shows the JAsCo bean comp1. All methods of comp1 are equipped with traps. As a result, whenever a method is called, its execution is deferred to the connector registry. The main method of communication of Java Beans is event posting, so firing an event also reschedules execution to the connector registry. When a trap is reached, the connector registry looks up all connectors that registered for that particular method or event. The connector on its turn dispatches to the hooks that have been instantiated with the corresponding method or event.

The main advantage of this trapped component model consists of the portability of the approach. JAsCo does not depend on a specialized virtual machine nor on some custom interfaces only available at certain systems. For example, a run-time environment optimized for embedded systems (JAsCoME) and an implementation of JAsCo for the .NET platform have been recently

proposed [25]. The drawback is of course that a performance overhead is experienced for all these traps, even if no aspects are applied.

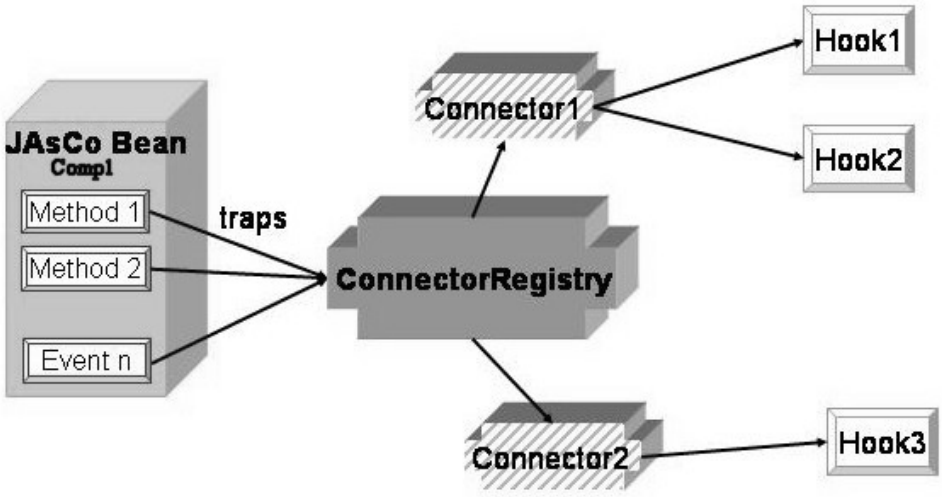

Figure 3. JAsCo run-time architecture.

2.3 JAsCo technology

JAsCo is a dynamic aspect-oriented approach, meaning that new connectors can be added dynamically and obsolete connectors can be removed. When adding or removing a connector, all instantiated hooks are added or removed. JAsCo is one of the most dynamic approaches currently available and offers the following features:

2.3.1 Central connector registry

JAsCo employs a central connector registry that contains all connectors and aspect beans. Without such a central registry, dynamically adding or removing aspects is not flexible at all as one has to iterate over all applicable object instances.

2.3.2 Remotely adding/removing aspects

JAsCo includes a very easy system for remotely (from outside the application) adding a connector. At regular time intervals, JAsCo scans the classpath[1] for new connectors. When a new connector has been found, it is automatically loaded in the system. As such, activating a connector in an application simply means placing the connector in the classpath of the application. Likewise, the removal of a connector is detected by the JAsCo run-time infrastructure and the connector and its instantiated aspect hooks are automatically removed from the system. Obviously, this system can be disabled if the performance penalty of scanning the classpath is considered too costly. The JAsCo system also offers an API for adding and removing aspects dynamically.

2.3.3 Precedence Strategies

Connectors are able to specify precedence strategies that define the priority between advices. In addition, the precedence is able to vary for the different advice types. Figure 4 illustrates a connector that instantiates two hooks: *logger* and *lock*. For the before advices, the behavior of *lock* needs to be triggered first, while for the replace advices, *logger* needs to be triggered first.

[1] It is also possible to specify a connector loadpath where JAsCo has to search for connectors.

```
1  connector Precedence {
2    // hook instantiations...
3
4    lock.before();
5    logger.before();
6    logger.replace();
7    lock.replace ();
8 }
```

Figure 4. Precedence strategy in a connector.

2.3.4 Combination Strategies

Precedence strategies are a solution to some feature interaction problems [26], however other combinations of aspects require a more expressive way of declaring how they cooperate. Therefore, extensible combination strategies are introduced. Combination strategies are implemented using regular Java and are instantiated in a connector. They are able to filter the list of applicable hooks of this connector on a per joinpoint basis. In addition, combination strategies are able to alter the priority and properties of the applicable hooks. Combination strategies are invoked for each execution of the applicable joinpoints. As such, they are able to dynamically influence the combined aspectual behavior. Suppose that an e-commerce system contains two discount aspects, a Birthday discount and a Frequent Customer Discount. Both discounts can however not be accumulated. A combination strategy is able to specify such behavior by removing one of the two discounts when they are both applicable for the joinpoint at hand. Figure 5 illustrates the instantiation of this exclusion combination strategy in a connector and the combination strategy itself.

```
1  connector DiscountConnector {
2    Discounts.Birthday birthday = new ...
3    Discounts.Frequent frequent = new ...
4
5    addCombinationStrategy(new
6      ExcludeCombinationStrategy(birthday,
7        frequent));
8 }

1  class ExcludeCombinationStrategy implements
2    CombinationStrategy {
3    private Object hookA, hookB;
4    ExcludeCombinationStrategy(Object a,Object b) {
5      hookA = a; hookB = b;
6    }
7    HookList validateCombinations(Hooklist list) {
8     if (list.contains(hookA)) {
9       list.remove(hookB);
10     }
11     return list;
12   }
13 }
```

Figure 5. An exclusion combination strategy.

2.3.5 Applying aspects on instances

It is possible to attach hooks onto specific object instances only, instead of all instances of a particular component type. The concrete instances that are subject of aspect application can be dynamically altered by employing the connector API.

2.3.6 Dynamic wildcard matching

JAsCo also supports the instantiation of a hook on expressions that contain wildcards. These limited regular expressions are matched at run-time. Consequently, when a new component is added to an application, it is automatically affected by all aspects that are instantiated using wildcards.

3 Jutta

3.1 Motivation

All dynamic features offered by JAsCo however induce a substantial run-time overhead. The overhead of JAsCo in real-life applications is often more than 1000% in comparison to hard-coding the advices, which is unacceptable. Notice that JAsCo is still in a prototype phase and therefore not a lot of attention has been paid to performance optimizations. The high overhead is mainly caused by the fact that the entire JAsCo run-time infrastructure is an aspect interpreter. For each joinpoint, JAsCo evaluates which hooks are applicable. When no connectors are added or removed, the set of applicable hooks remains unchanged for every joinpoint. As such, when the same joinpoint is encountered several times, the same logic for finding the appropriate hooks and executing their behavior is computed over and over again. Therefore, a huge performance gain can be realized when the combined aspectual behavior could somehow be compiled and cached for joinpoints that are encountered often. Of course, this compilation process requires some time, but when a joinpoint is encountered a lot, this pays off. In fact, this strategy is similar to just-in-time compilers used in modern virtual machines and therefore our approach is named *Jutta* (Just-in-time combined aspect compilation).

3.2 Jutta basics

The Jutta system allows generating and caching a highly optimized code fragment for a given joinpoint. This code fragment directly executes the appropriate advices on the applicable hooks in the sequence defined in the connector. As such, the system avoids iterating over all connectors and its hooks in order to find out which aspectual behavior is applicable. Rearranging the sequence of all applicable hooks for different advice types in order to implement precedence strategies is also avoided. Figure 6 illustrates the simplified Java counterpart of an example cached joinpoint behavior execution. The code fragment first initializes all applicable hooks with the current joinpoint and then executes only those advices that are defined in the connector in the correct sequence.

```
1  public void executeJoinpoint(Joinpoint jp) {
2    hook0._Jasco_initialize(jp);
3    hook1._Jasco_initialize(jp);
4    hook2._Jasco_initialize(jp);
5    hook1.before();
6    hook2.before();
7    hook0.replace();
8  }
```

Figure 6. Simplified Java counterpart of the cached combined aspectual behavior at a joinpoint.

The current implementation employs the Javassist [4] byte-code manipulation library in order to generate a combined hook behavior code fragment. Using Javassist, a java byte code class representation is generated on the fly, without requiring a compilation step. The overhead of generating a combined hook behavior code fragment is around 10ms on out test system[1]. The optimized code fragment is however only generated when the joinpoint is encountered the first time. As such, for joinpoints that are not executed, no overhead is experienced. The Jutta system also stores all code fragments generated for a given hook combination. As such, when the same hook combination is applicable to a different joinpoint, the overhead for generating the combined hook behavior code fragment is avoided. In addition, the Jutta system includes a set of pre-defined typical combined aspect behaviors. For those combined aspectual behaviors, the generation overhead is also avoided.

The JAsCo approach is however a dynamic AOP approach. As such, the cached behavior for a given joinpoint might become invalid. This happens when a connector is added that instantiates a hook that is applicable on the joinpoint or when a connector is removed that contains an applicable hook for the joinpoint. In addition, it is possible to change some properties of a connector dynamically so that the applicable context of the instantiated hooks is altered. The Jutta system has to be able to cope with these issues.

3.3 Hooks depending on dynamic values

Caching combined aspect behavior is not always achievable because it is possible that whether a hook is applicable or not, has to be re-evaluated for every execution of a given joinpoint. For example, when a hook defines a *cflow* condition in its constructor, this constructor has to be re-evaluated for every execution of a joinpoint. However, the entire constructor does not have to be re-evaluated. In this case, only the result of the *cflow* condition is able to change for different executions of the joinpoint. As such, partial evaluation techniques can be used to cache a partially evaluated constructor. In addition, for the particular *cflow* construct, it is sometimes possible to statically analyze whether the condition might ever be true or not by examining the call graph of an application. This technique is elucidated in [20].

3.4 Combination strategies

In general, caching the result of the combined behavior of all combination strategies for a given joinpoint is not possible. A combination strategy might depend on dynamic values in order to compute the list of applicable hooks. As such, combination strategies have to be recomputed for each execution of a given joinpoint. Some combination strategies do however not depend on dynamic values and always render the same result for a given input set of hooks. As such, these combination strategies do not need to be recomputed for every execution of a joinpoint. There is however no way to automatically find out whether a combination strategy depends on dynamic values or not. Therefore, the empty interface *DoNotCache* is introduced. When a combination strategy does not implement this interface, it is defined to always return the same set of hooks in the same sequence for a given input set of hooks. As such, the combination strategy only needs to be executed once for every input set of hooks. When a combination strategy does implement the *DoNotCache* interface, it is never cached and thus always executed for each applicable joinpoint.

4 JAsCo HOTSWAP

The Jutta system allows optimizing the aspect interpretation part of JAsCo dynamic AOP. The interception part however is still very slow. Inserting traps at all methods causes a performance overhead for all those methods, even no aspects are applied. In order to optimize this interception

[1] Pentium4 2GHz, 256MB RAM, Mandrake Linux 9.2, Java 1.4.2

system, we propose to employ the HotSwap technology of Java. HotSwap is introduced since Java 1.4 and allows to dynamically replace the byte code of a loaded class. As such, it is possible to install traps just-in-time when a new aspect is added to the system.

4.1 Approach

The JAsCo hotswap implementation allows installing traps in only those methods that are subject to aspect application. When a new aspect is added, all the methods where the added aspect is applied upon, are hot-swapped at run-time with a trapped version. Because HotSwap does not allow to replace single methods, the complete class byte code is replaced with a version where the applicable methods are trapped. All other methods of the class however remain untouched. Likewise, the original method byte code is installed when the aspect is removed again and if no other aspect is applicable at the method at hand.

The JAsCo HotSwap system does not exclude the regular preprocessing approach for installing traps. Classes that are already equipped with traps using the preprocessor are never altered. As such, when certain classes are definitely affected by aspects, they can be preprocessed to avoid the hotswap overhead at run-time. Furthermore, on platforms where no hotswap virtual machine is available, the preprocessing approach can still be used. JAsCo thus combines the best of both worlds, highly portable through the preprocessing approach and very little overhead when HotSwap is available.

The main drawback of the HotSwap system is that the virtual machine needs to run in debugging mode. As such, a global overhead is experienced depending on the virtual machine implementation. With the introduction of full speed debugging by the newest Sun virtual machines, this overhead is negligible. However, it appears that on our current Linux virtual machine (Sun JDK 1.4.2_03), a substantial overhead for debugging is still experienced, whereas on the same virtual machine for Windows practically no difference is noticeable.

4.2 Implementation Issues

Implementing a HotSwap system for AOP is technically quite challenging. The first problem is that typical HotSwap implementations do not allow altering anything of a class besides the method bodies. In order to implement an efficient AOP system, several fields containing reflective data about the joinpoints contained in the class are however required. Therefore, a separate class containing all those fields is generated each time new traps are installed. As such, the JAsCo HotSwap implementation requires somewhat more memory at run-time than when traps are installed using the traditional preprocessor.

Another problem is that HotSwap only allows replacing classes that are already loaded. As such, when new classes are loaded, the JAsCo run-time infrastructure needs to be notified in order to insert traps at those methods where aspects are applied. The obvious way to realize this is by employing the Java Debugging Interface (JDI) as the virtual machine is already running in debugging mode anyway. Using JDI, an event is received each time a new class is loaded and JAsCo is able to add traps to the methods of this class if necessary. However, by merely setting this “class prepared” breakpoint, the complete application is slowed down by up to 40%! In order to avoid this overhead, another solution is required to receive class loading events. Therefore, JAsCo employs the Java HotSwap facility to hotswap the system class loader by an enhanced class loader that notifies the JAsCo run-time infrastructure whenever a new class is loaded. As such, the overhead for the “class prepared” breakpoint is avoided. Hot-swapping the class loader can however cause problems when dedicated class loaders are employed. Typical J2EE application servers [10] depend on a custom class loader system and interfering with this system might cause the application to fail. Therefore, the current JAsCo implementation offers both class loading interception strategies.

5 Performance Evaluation

In order to evaluate the performance of the JAsCo approach enhanced with HotSwap and Jutta, we compare it to several state-of-the-art dynamic AOP approaches. Apart from JAsCo, the following dynamic AOP approaches are tested and compared: JBoss/AOP [8], PROSE [18], JAC [16] and AspectWerkz [3]. Notice that this selection is not meant as a comprehensive overview of all existing dynamic AOP systems. We merely selected those systems because they are publicly available and seemed stable enough in our opinion. Nevertheless, this selection is a good overview of current dynamic AOP approaches.

We employ two benchmark applications: a benchmark shipped with the JAC distribution and the PacoSuite benchmark [27]. The JAC benchmark application is a synthetic benchmark that invokes a set of public methods with different method signatures and empty method body implementations. This benchmark allows to precisely measure the overhead per method execution for applying aspects. The PacoSuite benchmark is meant as an evaluation of the performance in a realistic and non-trivial application. PacoSuite is a visual component composition environment, which is composed out of 1202 classes containing 34465 lines of code. The PacoSuite benchmark reads an XML composition description from file, validates the composition using a set of finite automata algorithms and finally displays the composition.

The next section shortly discusses the ideas and underlying implementations of the AOP approaches that are used in our experiments. Afterwards, section 5.2 discusses the benchmark results when no aspects are applied. Finally, section 5.3 presents the benchmark results when aspects are applied.

5.1 Employed dynamic AOP approaches

When comparing their underlying implementation, JAC and JBoss/AOP are rather similar AOP-technologies. Both approaches make use of traps which are automatically inserted at load-time of the application making use of byte-code transformations. Although both AOP-technologies are quit similar, JBoss /AOP is primarily intended as an aspect-oriented extension for the JBOSS J2EE application server [11], whereas JAC is developed as an AOP-framework which can be used as an alternative for a J2EE application server.

AspectWerkz is meant as a lightweight dynamic AOP framework and also inserts traps at load-time. In addition to employing a customized classloader like JBoss/AOP and JAC, AspectWerkz allows to employ the Java HotSwap functionality in order to hotwsap the system classloader for a classloader that inserts traps. All three approaches however insert traps at load-time. JAC always inserts traps at all methods, while JBoss/AOP and AspectWerkz do only insert traps at classes where aspects are already applied. As such, these approaches are not very dynamic because aspects can only be inserted and removed at trapped methods. Luckily, JBoss/AOP allows specifying a range of classes that have to be trapped, regardless of whether there are aspects applied or not. Unfortunately, for AspectWerkz, this is not possible.

PROSE employs a very different approach to intercept the program's execution than the previous technologies. PROSE[1] exploits the Java Virtual Machine Debugging Interface (JVMDI). A dedicated execution monitor is deployed on top of the JVMDI, which allows capturing relevant execution events. Whenever an event is encountered where an aspect is applied upon, the corresponding advice is executed.

[1] In [17], a second generation PROSE implementation has been proposed, which employs a dedicated virtual machine instead of the debugging interface. Sadly enough, as far as we know, this improved version is not publicly available at this time.

5.2 Benchmarks without aspects

For the first experiment, both the JAC and PacoSuite benchmark application are run on our test system[1] making use of the AOP technologies mentioned above, but without any aspects being applied. This allows observing the pure overhead of running the benchmark applications making use the AOP technologies. Notice that both benchmarks first run their application a couple of times in order to allow the Java virtual machine to optimize the code. This also allows some of the approaches to install their corresponding traps (JAC, JBOSS/AOP, JAsCo and AspectWerkz) or to perform some additional optimizations themselves. The execution times of these warm-up runs are not considered in our benchmark results. For each AOP approach, the experiments are performed at least ten times such that the standard deviation was less than 1% for the JAC benchmark application and less than 5% for the PacoSuite benchmark application.

Table 1: Benchmarks without any aspects applied.

Without Aspects	*JAC benchmark*	*PacoSuite benchmark*
No AOP/AspectJ	14 ms	590 ms
JAsCo 0.4.5	14 ms	684 ms
JAC 0.11	154689 ms	-
PROSE 1.1.2	14 ms	708 ms
JBOSS/AOP 4.0	507 ms	657 ms
AspectWerkz 0.9 RC1	2651 ms	-

Table 1 illustrates the result of the first experiment. For the JAC benchmark, one million "direct" iterations are performed. For the PacoSuite benchmark, three "visual" iterations are executed. For JBoss/AOP and AspectWerkz, we made sure that traps are inserted at all methods because otherwise, they are not dynamic at all, as no aspects can be added onto methods that are not trapped. When no traps are inserted, the performance of JBoss/AOP and AspectWerkz is the same as the original application. As explained before, AspectWerkz does not allow specifying that traps have to be inserted, even if there are no aspects. Therefore, we apply an empty aspect to all methods and remove it before the benchmark starts. This way only the overhead of the traps remains. Because removing aspects in AspectWerkz means fetching all possible joinpoints by name, this is not straightforwardly achievable for the PacoSuite benchmark (1202 classes).

At first glance, JAC appears to have a rather big overhead for its own benchmark in comparison to the other AOP approaches. Its low performance is however mainly caused by the slowness of the Java Reflective API which is employed within the JAC implementation. Unfortunately, no JAC results are available for the PacoSuite benchmark, as we were not able to run this application correctly because of JAC code generation errors. Both PROSE and JAsCo perform best in the JAC benchmark as they do not require traps for every method. For the PacoSuite benchmark, the overhead of employing the debugging interface seems to be higher than the overhead of inserting traps at all methods, since JBoss/AOP outperforms PROSE and JAsCo. As already mentioned in section 4.1, this is probably due to less optimal debugging interface implementation on Linux. Nevertheless, inserting traps at all methods is a feasible approach as the performance overhead in a realistic application scenario is only around 10%.

5.3 Benchmarks with aspects

As a second experiment, one simple aspect is applied upon each public method defined within the JAC and PacoSuite benchmark application. This aspect describes an around advice that increases

[1]Pentium4 2 GHz, 256MB RAM, Mandrake Linux 9.2, Java 1.4.2

a counter each time it is executed. For this experiment, 100000 "direct" iterations are performed for the JAC benchmark and three "visual" iterations for the PacoSuite benchmark. This results in respectively 800000 encountered joinpoints for the JAC benchmark and 210400 encountered joinpoints for the PacoSuite benchmark application. AspectJ is employed as utopian performance reference. In addition, the performance of JAsCo without the Jutta system being activated is measured. Table 2 illustrates the results. Notice that for the JAC benchmark, only one tenth of the iterations are performed in comparison to the previous experiment (100000 versus 1000000), so the timings for the JAC benchmark of Table 1 and Table 2 cannot be directly compared.

Table 2: Benchmarks with one around advice applied.

One Around Aspect on all public methods	*JAC benchmark (800 000 joint points)*	*PacoSuite bench (210 400 joint points)*
AspectJ 1.1	29 ms	645 ms
JAsCo 0.4.5; no Jutta	424928 ms	473665 ms
JAsCo 0.4.5	279 ms	753 ms
JAC 0.11	17198 ms	-
PROSE 1.1.2	946112 ms[1]	-
JBOSS/AOP 4.0	956 ms	949 ms[2]
AspectWerkz 0.9 RC1	487 ms	3698 ms[2]

In this experiment, JAsCo clearly outperforms the other approaches. This is mainly the contribution of the Jutta system, which is able to cache the application of aspects such that this information does not need to be calculated each time a joint point is encountered. If the Jutta system is disabled, the performance of JAsCo is very slow and is easily outperformed by all other AOP approaches. Again we observe that JAC, and this time also PROSE, have a rather big overhead in comparison to the other AOP approaches. For PROSE, this big overhead can probably be contributed to the lack of an efficient implementation which is able to cache which aspects are applied on which specific joint points.

Table 3: Benchmarks with three around aspects.

Three Around Aspects on all public methods	*JAC benchmark (800 000 joint points)*	*Overhead per advice execution.*
AspectJ 1.1	93 ms	0.032 ns
JasCo 0.4.5	395 ms	0.159 ns
JBOSS/AOP 4.0	1075 ms	0.442 ns
AspectWerkz 0.9 RC1	927 ms	0.380 ns

In a third experiment, three around aspects are applied upon each public method defined within the JAC bench. As the JAC benchmark application contains 8 public methods, 24 aspect instances are active in the system at the same time. This experiment is mainly performed because caching combined aspect executions is one of the main strengths of the Jutta approach. Table 3 displays the results of this experiment. JAsCo again outperforms the other tested dynamic AOP

[1] PROSE does not support an around advice, so we employ a before advice instead.

[2] The actual results for JBOSS/AOP and AspectWerkz are 1093 ms and 4859 ms. Both approaches however also trap private methods. This leads to a higher performance overhead Therefore, the performance of public methods is computed from the overhead per around execution times the number of public methods. Notice that this is not an issue in the JAC benchmark because it only consists of public methods.

approaches. However, it seems that JBoss/AOP scales better because adding 23 aspects only increases the execution time for JBoss/AOP with 12% whereas for JAsCo a 35% performance hit is experienced.

In order to assess the performance gain of the JAsCo HotSwap implementation, a last experiment is conducted. One single around aspect is applied upon one specific method defined within the JAC benchmark application. In addition, each method of the JAC bench is made advisable for JBoss/AOP and AspectWerkz such that aspects can be added at run-time. Notice that this is not required for JAsCo as JAsCo is still able to insert traps in these methods using HotSwap. As illustrated by Table 4, the JAsCo HotSwap implementation improves greatly over the other dynamic AOP approaches as traps are only added at one of the eight methods. Also notice that even the optimized JAsCo system is more than 1000% slower that AspectJ. As such, dynamic AOP is still far behind statically weaved approaches performance-wise.

Table 4: Benchmark with one around aspect applied upon one specific method.

One Around Aspect	*JAC benchmark (100 000 joint points)*
AspectJ 1.1	2 ms
JAsCo 0.4.5	29 ms
JBOSS/AOP 4.0	891 ms
AspectWerkz 0.9 RC1	275 ms

As a final note, it should be mentioned that the last three experiments employ an aspect which is described making use of an around advice, as this is the only kind of advice that is supported by each AOP approach that was used within this performance assessment, except for PROSE. Similar to AspectJ however, JAsCo also provides an explicit before and after advice. Apart from the conceptual benefit of an explicit before/after construct, such an advice can be executed faster, as no around advice chain needs to be built up. In case of experiment two for instance, the performance of JAsCo for the JAC-benchmark application is improved by 15% if before advices are applied instead of around advices.

6 Related Work

Apart form the approaches employed in our benchmarks, several other AOP approaches are introduced for enabling dynamic AOP. Event based aspect oriented programming (EAOP) allows specifying crosscutting concerns by employing event patterns which are described using a formal language [6]. Because of this formal model, advanced detection and resolution of aspect interactions becomes possible. On the implementation level, EAOP inserts traps that query a central execution monitor, similar to the JAsCo connector registry. The execution monitor has a global view of the executing application and contains all active EAOP artifacts. In contrast to JAsCo, EAOP inserts traps by source-code transformations.

Using Caesar [14], an aspect is described in terms of an Aspect Collaboration Interface (ACI). Each concrete aspect needs to implement the required methods specified by its corresponding ACI. Aspect bindings connect the aspect implementations to different concrete deployment contexts. One of the major contributions of the Caesar approach is the introduction of aspectual polymorphism. Aspect bindings are able to implement a binding for different types and the concrete binding is resolved dynamically using the type of the object at hand. In this viewpoint, aspectual polymorphism is similar to the concept of late binding found in object oriented languages.

Filman [7] proposes dynamic injectors in order to introduce aspects within an application. These dynamic injectors are incorporated into the OIF (Object Infrastructure Framework), a

CORBA centered aspect-oriented system for distributed applications. Dynamic injectors are first class objects that can be added and adapted at run-time. At the implementation level, a wrapping approach is employed for injecting the logic of an aspect within a component communication channel.

Wool [19] is a dynamic AOP framework that supports two different dynamic weaving strategies. The Wool system employs the Java Debugging Interface to intercept the execution of the base program. In this respect, Wool is similar to the PROSE approach. However, aspects can also be inserted into the target joinpoints directly by employing Java HotSwap. The original contribution of Wool is that aspects are able to implement their own heuristics for deciding whether they are invasively inserted or not. The difference with the JAsCo hotswap implementation is that JAsCo only insert traps, not full advices. In Wool however, aspects lose their identity at run-time. In addition, Wool requires to hotswap more as for each additional aspect, the classes containing the applicable joinpoints need to be hot-swapped again.

Finally, AspectS [9] introduces dynamic AOP support within the Squeak/Smalltalk environment. Pointcuts and their corresponding advices are described making use of plain Smalltalk. By sending the install and uninstall message to an instance of such an aspect, aspects are activated and deactivated within the application at run-time. At the implementation level, AspectS makes use of the dynamic properties of Smalltalk itself. In this case, Method wrappers are used which are placed around a compiled method by replacing its entry in the method dictionary of a class. This way, it is possible to easily add behavior, in this case aspect advices, to method invocations.

7 Conclusions and Future Work

This paper presents the HotSwap and Jutta systems in order to improve the performance of JAsCo dynamic AOP. The performance evaluation clearly indicates that the JAsCo implementation enhanced with Jutta and HotSwap improves current state-of-the-art dynamic AOP. However, the overhead is still a lot larger than when statically weaved languages like AspectJ are employed.

The Jutta system is not only applicable to JAsCo. The ideas can be recuperated in any other dynamic AOP approach regardless of which technology is used for intercepting the program execution. Therefore, we plan to decouple the Jutta system from JAsCo and as such achieve a general dynamic AOP optimizer.

The HotSwap system is however only a short and medium term solution for intercepting the program execution. In the long term, the best approach to support dynamic AOP or even regular AOP consists of dedicated aspect-oriented virtual machines as for example proposed by PROSE2 [17]. Indeed, preprocessing, load-time trap insertion or employing the debugging interface of a virtual machine are all solutions that are feasible on the short-term, but are quite cumbersome and error-prone in comparison with a dedicated execution environment. The Jutta system and ideas are however still applicable for such dedicated virtual machines.

Acknowledgements

We owe our gratitude to Prof. Dr. Viviane Jonckers for her invaluable help during our research and for proof reading this paper. Since October 2000, Wim Vanderperren is supported by a doctoral scholarship from the Fund for Scientific Research (FWO or in Flemish: “Fonds voor Wetenschappelijk Onderzoek”). Davy Suvée is supported by a doctoral scholarship from the Flemish Institute for the Improvement of the Scientifical-Technological Research in the Industry (IWT or in flemish: ``Vlaams instituut voor de bevordering van het wetenschappelijk-technologisch onderzoek in de industrie").

References

[1] AspectJ Website: http://www.aspectj.org

[2] Baker, J. and Hsieh, W. *Runtime aspect weaving through metaprogramming.* In Proceedings of the first International Conference on Aspect-Oriented Software Development. Enschede, The Netherlands, April 2002.

[3] Bonér, J. and Vasseur A. *AspectWerkz: a dynamic, lightweight and high-performant AOP/AOSD framework for Java.* Available at: http://aspectwerkz.codehaus.org

[4] Chiba, S. and Nishizawa, M. *An Easy-to-Use Toolkit for Efficient Java Bytecode Translators.* In Proceedings of the second International Conference on Generative Programming and Component Engineering. Erfurt, Germany, September 2003.

[5] Cibran, M., D'Hondt, M. and Jonckers, V. *Aspect-Oriented Programming for Connecting Business Rules.* In Proceedings of the 6th International Conference on Business Information Systems. Colorado Springs, USA, June 2003.

[6] Douence, R., Motelet, O. and Südholt, M. *A formal definition of crosscuts.* In Proceedings of the 3rd International Conference on Reflection. Kyoto, Japan, September 2001.

[7] Filman, R.E. *Applying aspect-oriented programming to intelligent systems.* Position paper at the ECOOP 2000 workshop on Aspects and Dimensions of Concerns. Cannes, France, June 2000.

[8] Fleury, M and Reverbel, F. *The JBoss Extensible Server.* In Proceedings of Middleware 2003 Int Conference, Rio de Janeiro, Brazil, LNCS(2672), January 2003.

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

[10] Java J2EE website. http://java.sun.com/j2ee/

[11] JBOSS J2EE Application Server Website. http://www.jboss.org/

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin J. *Aspect-oriented programming.* In Proceedings of European Conference for Object-Oriented Programming. Jyväskylä, Finland, June 1997.

[13] Lieberherr, K., Lorenz, D. And Mezini, M. *Programming with Aspectual Components.* Technical Report, NU-CSS-99-01, March 1999. Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html.

[14] Mezini, M. and Ostermann, K. *Conquering Aspects with Caesar.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[15] Ossher, H. and P. Tarr, *Using multidimensional separation of concerns to (re)shape evolving software*. Communications of the ACM 44 (2001), pp. 43–50.

[16] Pawlak, R., Seinturier, L., Duchien, L. and Florin, G. *JAC: A flexible solution for aspect-oriented programming in Java.* In Proceedings of the third International Conference on Reflection. Kyoto, Japan, September 2001.

[17] Popovici, A., Alonso, G. and Gross, T. *Just-in-time aspects: efficient dynamic weaving for Java.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[18] Popovici, A., Gross, T. and Alonso, G. *Dynamic Weaving for Aspect-Oriented Programming.* In Proceedings of the 1st International Conference on Aspect-Oriented Software Development. Enschede, The Netherlands, April 2002.

[19] Sato, Y., Chiba, S. and Michiaki, T. *A Selective, Just-in-Time Aspect Weaver.* In Proceedings of the second International Conference on Generative Programming and Component Engineering. Erfurt, Germany, September 2003.

[20] Serini, D. and De Moor, O. *Static analysis of aspects.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[21] Suvee, D., Vanderperren, W. and Jonckers, V. *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[22] Vanderperren, W. *A pattern based approach to separate tangled concerns in component based development.* In Proceedings of ACP4IS workshop at AOSD 2002. Enschede, The Netherlands, April 2002.

[23] Vanhaute, B., De Win, B. and De Decker B. *Building Frameworks in AspectJ*. Workshop on Advanced Separation of Concerns.

[24] Verheecke, B., Cibran, M. A. and Jonckers, V. *AOP for Dynamic Configuration and Management of Web services in Client-Applications*. In Proceedings of 2003 International Conference on Web Services. Erfurt, Germany, September 2003.

[25] Verspecht, D., Vanderperren, W., Suvee, D. and Jonckers, V. *JAsCo.NET: Unraveling Crosscutting Concerns in .NET Web Services.* In Proceedings of Second Nordic Conference on Web Services NCWS'03. Vaxjo, Sweden, November 2003.

[26] Workshop on “feature interaction in composed systems” at ECOOP 2001. Program available at http://www.info.uni-karlsruhe.de/pulvermu~/workshops/ecoop2001.

[27] Wydaeghe, B. and Vanderperren, W. *Visual Component Composition Using Composition Patterns.* In Proceedings of Tools 2001. Santa Barbara, USA , July 2001.

Dynamic AOP and Runtime Weaving for Java — How does AspectWerkz Address It?

Alexandre Vasseur
avasseur@bea.com

Abstract

This paper describes how AspectWerkz [1]'s dynamic AOP capabilities for Java [2] are extended to support runtime weaving. While static weaving (post compilation or class load time) allows for some dynamic AOP features with the help of a join point centric weaving model and an aspect container component, runtime weaving is richer in the sense that weaver targets can be declared at runtime. A working implementation allows to define key principles for enabling of dynamic AOP with pointcut redefinition: "in process HotSwap" and "two phases weaving."

Keywords

Aspect-Oriented Programming, AOP, AOSD, dynamic AOP, runtime weaving, HotSwap

1 Introduction

One central component in AOP is the weaver. Given a set of target programs and a set of defined aspects, the weaver alters the target program to weave in the aspects capabilities, embodied by advices, mixins and binding rules.

In the Java [2] landscape the most common approach is to alter the compiled class bytecode to produce a bytecode where aspects are weaved in. If bytecode manipulation is easy to do through a post compilation phase, it is a bit more complex to provide a solution generic enough to apply bytecode modification at class load time, but some projects like JMangler [3] and AspectWerkz have achieved it, and Java 1.5 will standardize this approach with the JSR-163 [4].

Allowing dynamic AOP on top of such a static weaving phase relies on internal AOP constructs. AspectWerkz allows many dynamic AOP operations — like adding an advice or changing an introduction implementation — with the only requirement that the pointcut is already defined. It is then possible to rearrange advices and swap mixin implementations at runtime, without any class reloading or new weaving phase. The first part of this paper aims at explaining some of the major implementation details needed to enable dynamic AOP constructs.

Even if this approach is suitable for many use-cases, one missing feature is to allow pointcut definition at runtime, so that a target component can have its normal behavior (eventually with aspects) up to a point where it is altered with new AOP constructs on totally new join points. This allows for new constructs to be applied without any prior overhead. Instead of preparing each method call, execution or field access with a join point, pointcuts are defined at runtime and corresponding targets are altered at runtime to weave the new aspect constructs only when it is required.

The second part describes the proposed solution and the key components that allow runtime definition and redefinition of pointcuts without class reloading, and with a prior runtime performance overhead reduced to zero.

2 Static weaving and dynamic AOP constructs

Dynamic AOP constructs can be built on top of a classic static weaver, where target classes are altered once, whether in a post compilation phase prior to deployment, or at class load time thus at deployment time.

A join point centric weaving model can provide enough redirection level in the weaved bytecode to be able to alter AOP constructs at the underlying framework level.

The AspectWerkz dynamic AOP framework follows such a design since its early conception.

2.1 Static weaving

The common approach for applying AOP principles in the Java landscape is bytecode manipulation. Several APIs like BCEL [5], Javassist [6] and ASM [7] provide access to the bytecode instructions and several Java AOP frameworks like AspectJ [8] and AspectWerkz [1] make use of such facilities. The bytecode manipulation is a way to support a fine-grained join point model in different context such as field access and method calls and executions.

Static weaving can be defined as the single phase operation that given a class representation and a set of aspects alters the representation to produce a new class where aspects are weaved in at precise join points.

The first weavers for Java AOP only supported static weaving through post compilation tools, and some recent steps [3][1] ahead demonstrate that the same operation can be achieved at class load time no matter the target environment. AspectWerkz's hooking architecture supports integration in J2EE application servers with complex class loading schemes, non-standard JREs like IBM and non-standard JVMs like BEA JRockit. AspectWerkz extends a concept that has been first defined by the JMangler [3] project. The class load time weaving will be standardized in Java 1.5 through the JSR-163[1] [4]. This feature can be used for AOP but is there to support better profiling as an enhancement of JPDA [9].

Once statically weaved (at compilation time or at class load time), the bytecode is loaded in the JVM and the target weaved *java.lang.Class* object is defined. The application then runs, without any further modifications at bytecode level.

Dynamic AOP constructs are then enabled through a well-designed weaving model and aspect container [10].

2.2 Weaving model and aspect container

The weaving model defines the way the target class bytecode is altered when a pointcut is matched.

The aspect container [10] is in charge of managing deployed aspects and mixin implementations, and supports fine deployment options (perThread [11] aspect shared between target instances on a per execution thread basis) as well as advice rearrangement.

Both are key elements to allow dynamic AOP constructs. If the weaving model does not have enough indirection level, it will not be possible for example to add new advices at the join point.

The AspectWerkz's weaving model provides a good level of indirection by being *join point centric*. The altered bytecode contains information about the join point only, and the aspect

[1] Java 1.3 and 1.4 supported "bytecode loaded" event callbacks at C JVMPI level, but without providing the defining ClassLoader instance, which appears to be a requirements in AspectWerkz AOP. Java 1.5 fixes this and also provides a Java level API, such the one already provided in BEA JRockit since v7 (Java 1.3).

container then keeps information about which advice(s) is (are) associated with the join point, or which mixin implementation(s) is (are) associated with the weaved in mixin interface(s).

At runtime, a join point manager allows to invoke the current advice(s) bounded at the join point. Dynamic AOP constructs around the join point are thus possible at aspect container level without any new bytecode modifications.

Some examples of the weaving model allows for a better understanding. The examples are provided as source code for the sake of simplicity as compared to bytecode excerpt.

2.2.1 Method execution pointcut

Given the initial bytecode representing:

```
public class Target {

    public Object doSomething(int param) {
        // ...
        // doSomething original method body
        return result;
    }
}
```

And given a method execution pointcut matching *Target.doSomething(int)*, the static weaving in AspectWerkz leads to the following representation:

```
public class Target {

    private static __clazz = Class.forName("Target");

    private static JoinPointManager __jpManager =
JoinPointManager.get(__clazz);

    private Object __doSomething(int param) {
        // ...
        // doSomething original method body
        return result;
    }

    public Object doSomething(int param) {
        __jpManager.proceedWithExecutionJoinPoint(
                -0x25ca1875,//method hash
                //... join point information
                );
    }
 }
```

The original method is renamed, and an indirection level is added as a replacement through an added method with the original method name and the same signature. No explicit call to the original method appears since the join point manager will handle it.

No information about the bounded advices is weaved in into the target class. Only a join point manager instance is created as a static field, providing enough information when a join point is reached to determine to which class, instance and original method execution it relates. A method hash (representing an integer) is defined, so that direct access through an array lookup can be performed at runtime. The method hash is computed given the original method signature.

The *proceedWithExecutionJoinPoint(..)* call will invoke all bounded advice(s) (if any), providing support for before, around and after advices.

For a static method, the weaving model is exactly the same.

2.2.2 Caller side pointcut

Given the initial bytecode representing:

```
public class Caller {

    public Object callTarget(Target target) {
        // ... body
        Object result = target.doSomething(...);
        return result;
    }
}
```

And given a caller side method pointcut matching "*Caller calls Target.doSomething(..),*" the static weaving in AspectWerkz leads to the following representation:

```
public class Caller {

    private static __clazz = Class.forName("Target");

    private static JoinPointManager __jpManager =

JoinPointManager.get(__clazz);

    public Object callTarget(Target target) {
        // ... body
        // altered body below
        Object result = __jpManager.proceedWithCallJoinPoint(
                0xe30193c,//called method hash
                //... join point information
                );
        return result;
    }
}
```

The weaving model for the caller side pointcut construct does not express the advices bounded. This is done at the aspect container level and invoked at runtime through the join point manager. The before, around and after advices will be called subsequently.

The same approach is used for field set and get pointcuts.

2.2.3 Mixin

Mixins provide ability to add new methods to a class. A first simplistic approach would be to weave in the mixin implementation bytecode in the target class, but this would not be suitable for dynamic AOP constructs.

Again, AspectWerkz's weaver adds an indirection level. In this case there is no join point or join point manager involved, but a mixin manager, which is a subpart of the aspect container.

Given the following mixin implementation that implements the user provided interface *MixinIntf*:

```
public interface MixinIntf {

    public void sayHello();

}
```

```
public class MixinImpl implements MixinIntf {

    public void sayHello() {
        // ...
        // sayHello implementation
    }

}
```

And given that the mixin is defined to be bounded on "all class(es) whose name is Target," the weaved bytecode will be represented as follows:

```
public class Target implements MixinIntf {

    private static __clazz = Class.forName("Target");

    private static AspectManager __aspectManager =
AspectManager.get(__clazz);

    public void sayHello() {
        __aspectManager.getMixin(1).//mixin index
                ___AW_invokeMixin(
                        1,//mixin method index
                        //...mixin invocation parameters
                        );

    }

    // ...
    //
    // might depends on other AOP construct
    // applied to Target class

}
```

2.3 Dynamic AOP constructs

The indirection level brought by the join point centric static weaving model, and a mixin / aspect container provide the low level mechanisms to apply several dynamic AOP constructs in AspectWerkz. Since the join points have been defined during the static weaving phase, the dynamic AOP constructs do not rely on a new weaving phase.

At join point first execution, the aspect container gathers information about the join point and is then able to determine which advices are bounded. At the first mixin invocation, the mixin is instantiated. Deployment models [11] like *perThread* are implemented at the container level using the Prototype pattern [12].

At runtime, it is then possible to perform dynamic AOP operation on the system through the aspect container. An API allows to lookup aspects or pointcuts (based on their name or on specific method and class matching) to:

- add an advice at a specific pointcut,
- remove an advice at a specific pointcut,
- reorder advices at a specific pointcut,
- swap the implementation of a mixin,

- add an aspect providing the new aspect does not define new pointcuts, and defines advices (represented as regular java methods in AspectWerkz) that will be programmatically bounded to existing pointcut(s).

The following demonstrates how to use the API to access an existing pointcut and add a new advice based on the aspect, the pointcut and the advice names:

```
ExecutionPointcut executionPointcut = SystemLoader.
        getSystem("namespace").
        getAspectManager().
        getPointcutManager("aspectName").
        getExecutionPointcuts(classMeta, methodMeta).
        get("pointcutName");

executionPointcut.addAroundAdvice("Aspect.newAdvice");
```

The following demonstrates hot to use the API to replace a mixin implementation, based on the introduction and the new implementation names:

```
SystemLoader.getSystem("namespace").
        getAspectManager().
        getMixin("introductionName").
        ___AW_swapImplementation("MixinOtherImpl");
```

It is possible to call those operations from within an advice itself, which allows for adaptive behavior.

2.4 Limitations

AspectWerkz's static weaving model and aspect container are key elements to enable dynamic AOP constructs.

A shortcoming in this implementation is that the join point has to exist as a result of the static weaving phase, no matter if it is through post compilation or through class load time weaving. It means that runtime definition of new pointcuts is not possible.

One approach would be to have very generic pointcuts, and then adapt the advice's behavior according to runtime type information available in the join point. If such rules can be applied elegantly with *cflow* constructs like it is supported in AspectJ [8] and AspectWerkz [1], it might lead to a global overhead, both at weave time and at runtime.

A key requirement in dynamic AOP appears to be the capability to add new pointcuts in the running system, with a good control over the performance overhead before the pointcuts are applied, so that almost all classes in the JVM can be altered at runtime without prior preparation and without any overhead.

Wool [13] tries to address this issue using a hybrid approach between bytecode weaving at runtime and debugging breakpoints usage. Axon [14] addresses it by using a JVMDI [15] event callback registration to be notified at each method execution and do the necessary advice calls — if any.

The following part explains the implementation done in AspectWerkz to allow runtime pointcut definition while still being compliant with the classic static weaving approach.

3 Runtime weaving

Runtime weaving has actually been supported in the Java landscape since Java 1.4 and the HotSwap facilities of JVMDI [15] that allows redefinition of the class bytecode at runtime, without reloading the class.

Even if hot deployment is a common pattern to update a system, we don't think it solves the problem. Re-deploying an application in an application server is not a seamless operation and affects the application uptime. Runtime weaving allows much more fine-grained level control.

HotSwapping can indeed be used to reweave target classes during runtime, upon user request or system request. Use of this operation can allow adding pointcuts at runtime and thus address a missing core capability in AspectWerkz's static weaving model for dynamic AOP.

A first implementation has been done to validate the viability of such an approach, and to point out main issues and components, as well as to ensure that performance overhead prior pointcut runtime definition is minimal.

The proposed solution relies on a hybrid approach between static weaving and runtime weaving as well as on an in-process HotSwap custom API. This results in a null runtime overhead prior pointcut addition, at the cost of a small load time overhead.

3.1 HotSwap in Java

HotSwap, also called class redefinition, is a feature added in Java 1.4. It is part of the JVMDI [15] API available at C level and remotely at Java level through JDWP [16].

When JVMDI is activated through the *–Xdebug* flag when launching a Java 1 .4 HotSwap compliant JVM, the method call dependencies are recorded internally.

The HotSwap API allows to submit new bytecode for an already loaded class in the running JVM. Former methods that appeared to have changed are relinked in the JVM internal representation based on the method call dependencies recorded so far. When a new invocation is done, it goes through the new method, part of the HotSwapped class.

Even though the HotSwap API looks very attractive, we think there are potential problems. The current Java directions seem to aim at having HotSwap more usable:

- The HotSwap API has not been adopted by all JVMs. BEA JRockit does not support it yet. Since the JSR-163 [4] part of Java 1.5 redefines the API, we can expect changes as regards HotSwap support.
- The HotSwap API in Java 1.4 was part of JVMDI thus required the use of the *–Xdebug* flag. Such a requirement can reduce the JVM performance, especially when running in server mode. In Java 1.5, it seems it won't be required anymore, but the effective overhead will still have to be measured.
- The HotSwap API can only be called at Java level through a JDWP connector, thus from a second JVM (as done in Wool [13]), or only at C level from within the target JVM. Java 1.5 will bring the API at Java level within the target JVM, and will add the "in-process HotSwap" feature as a default.

One key issue that remains when using HotSwap is that the class representation represented by the new bytecode has to conform to some restricting rules:

- The class schema must not change: there is no way to add new methods, even private, or to change signatures of the methods. No field can be added.
- The class initialization (<clinit>) is not rerunned.

In theory the API allows to support schema changes, but up to now we are not aware of such JVMs.

Obviously, the previously described weaving model of AspectWerkz has to be adapted to be HotSwap compliant due to the current schema change restriction. The next parts express the required changes.

3.2 In-process HotSwap

In the AspectWerkz's runtime weaving implementation we decided to use the HotSwap API directly from within the target JVM, without using a JDWP connection. We think it is a way to validate what will be standardized in Java 1.5.

Since in Java 1.4 the API is only available at C level we have set up a Java API on top of it, using the JNI [17] features. It allows to have in Java 1.4 an API that is almost the same that the one defined by JSR-163, and that simplifies the use of the HotSwap feature by bringing "in process HotSwap" Java level API.

```
public class InProcessHotSwap {
     static {
        System.loadLibrary("aspectwerkz");
    }

    private static native int hotswap(
            String className,
            Class orginalClass,
            byte[] newBytes,
            int newLength
            );

   // … utility methods

}
```

The target JVM has to activate the HotSwap using the *–Xdebug* flag and having the system dependant built JNI based in process HotSwap library in its path.

3.3 Two phases weaving

One key issue to solve when using HotSwap is that we are for now restricted to a set of bytecode changes that do not lead to a "class schema change," due to current JVM limitations.

As detailed in the previous part this is almost always the default weaving model in AspectWerkz, excepted for execution pointcut that leads to addition of methods.

To address these changes while still being compatible with a class load time weaving model, we decided to use a hybrid approach where target classes are weaved at class load time as required, prepared for further HotSwap if needed, and then activated through HotSwap when new pointcuts are defined.

This hybrid approach thus makes use of both class load time weaving and runtime weaving.

3.3.1 Class load time preparation

To address the current JVMs schema change limitation, a class preparation phase is required. We decided to allow fine grained control on this preparation phase, thought it can be enable for all loaded classes if required. The classes to prepare are thus explicitly declared in the AspectWerkz's XML based descriptor.

```
<?xml version="1.0"?>
<aspectwerkz>

    <system id="demo">
        <prepare class="Target" />
        <prepare package="com.service.*" />

        <!-- ... -->
        <!-- other AspectWerkz related elements -->
```

```
    </system>
</aspectwerkz>
```

When a loaded class matches those "<*prepare* …>" declarations, the following transformations are applied:

- add a private static *__clazz* field to reference the current Class object
- add a private static *__jpManager* field to reference the current Class join point manager
- for all methods, add a new empty method whose name is the AspectWerkz's prefixed method name, if and only if the method is not already matched by a declared pointcut

The added methods are empty, simply returning the default value required for JVM bytecode compatibility.

The preparation phase is fully compatible with the static weaving phase. A prepared class will be eligible for runtime weaving while still being affected during the static weaving phase by declared AOP constructs (if any).

The following illustrates the change if the class *Target* presented so far is prepared:

```
public class Target {

    private static __clazz = Class.forName("Target");

    private static JoinPointManager __jpManager =
JoinPointManager.get(__clazz);

    private Object __doSomething(int param) {
        // empty method for further HotSwap
        return null;
    }

    public Object doSomething(int param) {
        // unchanged.
        // doSomething original method body
        return result;
    }

    // other class methods and constructors
    // advised if required during the static weaving phase

}
```

3.3.2 Activation phase

The activation phase allows the triggering of a new bytecode transformation based on the bytecode obtained after the load time preparation.

The class bytecode is weaved so that when an unaffected method match a newly defined execution pointcut, the original method body is moved into the empty method added in the preparation phase while the original method body receives the join point centric weaving model code.

The class schema is not changed and this transformation can thus be submitted to the in process HotSwap API with the current JVM limitations. The resulting bytecode is the same as the one obtained in a static weaving, where the added join points would have been initially defined.

After the activation phase the runtime overhead is the one that occurs when a method is advised, but prior to this the runtime overhead is absolutely null since the exact original bytecode gets executed with the same stack trace.

3.4 Caller side constructs

Caller side constructs — like field get set and method call pointcut — are easier to be adapted for two phase weaving and HotSwap requirements since they do not require a schema change.

During the activation phase, the affected method body is changed to insert the call to the join point manager. This change is compatible with HotSwap schema changes restrictions providing that the join point manager reference has been added during the preparation phase.

4 Future work

This AspectWerkz's implementation for runtime weaving allowed us to define and validate a way to allow join point runtime redefinition without any prior runtime overhead for non advised method calls and executions and non advised field accesses. The two phase weaving approach, based on an in process HotSwap Java level API allows to address this major dynamic AOP requirement, whereas a join point centric weaving model allows to address advice and aspect rearrangement on existing join points.

One key element in this hybrid model is the prepare phase overhead. This one has to be as low as possible so that many classes — if not all — can be declared as to be prepared. The preparation phase is indeed very lightweight, and consists only in adding empty methods that won't be called prior activation and two fields to a class bytecode.

The hybrid approach allowed us to address a limitation introduced by the HotSwap API schema changes restriction. This could be simplified if JVMs would support unrestricted schema changes during HotSwap.

A more important issue that will have to be addressed in order to enhance the current implementation is that once a class is loaded in the JVM, there is currently no API to retrieve its current bytecode representation. The Java technology provides only a way to retrieve the original bytecode, as it is stored in the class path (eventually remotely), or to retrieve a single method bytecode representation (using JVMDI [15]). We think this limitation might be a major issue for large HotSwap based systems, as well as systems making use of several bytecode transformation sub-systems like AOP, or profilers. The current implementation makes use of a local cache, which is enough to have it work for small applications. This issue may come from the historical use-case that required HotSwap: runtime debugging facilities based on source code changes. We think this issue can be addressed at the JVM level, by providing an API that would allow to extract the exact bytecode representation of a class as it is actually at runtime, and not as it is when loaded.

JVM level capabilities is becoming a crucial success factor for complete dynamic AOP solutions, and the border between what should be handled by the JVM and what should be handled by the AOP framework has yet to be defined. The work described in this paper aimed at having a working solution today to allow pointcut redefinition with Java 1.4. Java 1.5 will even more ease this approach since in process HotSwap (with a java level API) and class load time weaving hook are standardized. As a future work, we will look at how aspect hot deployment can trigger runtime weaving.

No matter how dynamic AOP will be technically achieved in Java, we will have to address complex use-case in this area: how to allow for good system state control and management, how can I proceed to reproduce a problem in QA environments when several external events might have modified the running system?

Acknowledgments

Many thanks to Jonas Bonér, founder of AspectWerkz, for his invaluable feedback on how to improve the hybrid approach as well as on this paper.

References

[1] Bonér, J. AspectWerkz: dynamic AOP for Java. 2003. http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf

[2] Gosling, J., Joy, B., Steele, G. The Java Language Specification (2nd edition) Addison-Wesley, 2000.

[3] Kniesel G., Costanza P, Austermann M. JMangler — A Framework for Load-Time Transformation of Java Class Files. IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001).

[4] JSR-163. At http://www.jcp.org/en/jsr/detail?id=163, 2004.

[5] BCEL, The Byte Code Engineering Library. http://jakarta.apache.org/bcel/

[6] Javassist, Java Programming Assistant, http://www.csg.is.titech.ac.jp/~chiba/javassist/

[7] ASM, http://asm.objectweb.org/

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting Started with AspectJ. Communications of the ACM, 44(10):59–65, October 2001.

[9] JPDA, Java Platform Debugger Architecture, http://java.sun.com/products/jpda/

[10] Bonér J., What are the key issues for commercial AOP use — how does AspectWerkz address them ?. Submitted for AOSD 2004.

[11] AspectWerkz homepage, http://aspectwerkz.codehaus.org, 2004.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] Shigeru Chiba, Yoshiki Sato, Michiaki Tatsubori. Using HotSwap for Implementing Dynamic AOP Systems. ECOOP'03 workshop on Advancing the State of the Art in Runtime Inspection (ASARTI), 2003.

[14] Aussmann S., Haupt M. Axon — Dynamic AOP through Runtime Inspection and Monitoring. ECOOP'03 workshop on Advancing the State of the Art in Runtime Inspection (ASARTI), 2003.

[15] JVMDI, The Java VM Debug Interface. http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jvmdi-spec.html

[16] JDWP, Java Debug Wire Protocol. http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdwp-spec.html

[17] JNI, Java Native Interface. http://java.sun.com/j2se/1.4.2/docs/guide/jni/

Dynamic Aspects for Web Service Management

Bart Verheecke
María Agustina Cibrán

Vrije Universiteit Brussel
{Bart.Verheecke, Maria.Cibran}@vub.ac.be

Abstract

We observe that current approaches for the integration of web services hard-wire the service references into client applications, affecting adaptability and reusability. Moreover, support for client-side management is hardly provided. To enable the development of more flexible and robust applications we propose the Web Services Management Layer (WSML). In this paper we identify different aspects in the WSML and show why dynamic Aspect Oriented Programming (AOP) is required to realize the core functionality of the WSML. As a proof-of-concept we present an implementation of the WSML realized in JAsCo, a dynamic AOP language, which enables runtime pluggability of aspects.

1 Introduction

Web services are modular applications that are published, localised and invoked over a network, typically the Internet, by means of W3C standards such as SOAP [1], WSDL [2] and UDDI [3]. However, the current approaches typically used to integrate services in client applications are rather static. State-of-the-art tools like MS Visual Studio.NET and BEA WebLogic adopt the Wrapper Approach: a client-side proxy is generated for each web service. Programmers can invoke web methods on this class and do not have to take into account that they are actually dealing with remote procedure calls (RPCs). By treating the services as regular software components, the specific requirements of web services are completely ignored. Services are organisationally fragmentized, can be asynchronous and latent, can become unavailable due to unpredictable network conditions and thus require more overall management [4]. As a consequence applications result unmanageable and not adaptable to changes in the business environment [5].

To avoid this hardwiring we propose the *Web Services Management Layer* (WSML) [6,7]. In the WSML we identify the need for *Aspect Oriented Programming* (AOP). Furthermore, because of the volatile nature of the service environment, a dynamic AOP technology is needed to realize the core functionality of the WSML.

Section 2 presents our approach and in section 3 the identified dynamic aspects are explained. Section 4 shows the implementation of the WSML using *JAsCo* [8,9], a dynamic AOP language, as a proof-of-concept of our ideas. Related work is described in Section 5. Finally, we conclude and present some future work in Section 6.

2 Requirements for the WSML

We propose the Web Service Management Layer (WSML) a platform that allows easy integration and client-side management of web services in client applications. Figure 1 shows the role of the WSML in a service-oriented application. The main objectives pursued by the WSML are:

- *The ability to dynamically select the services to integrate*: As a consequence of hard-wired references it is impossible to dynamically swap to other services that better accommodate to the application requirements or to hot-swap to other available services when the currently integrated ones become unavailable. Moreover it is unfeasible to dynamically integrate new services that were not known or anticipated at development time. The WSML tackles this issue making it possible to select and integrate services at run-time.
- *The consideration of non-functional properties in the selection of services:* Another WSML objective is to consider non-functional properties and Quality of Service constraints (QoS) to guide the selection of the most appropriate services. Selection criteria can be based on properties defined in the services descriptions and that can be retrieved and checked at the moment the criteria are applied (e.g. price, distance). Another possibility is that the properties involved in the selection criteria are not anticipated and defined in the services. These properties depend on the behaviour of the service at run-time. Examples of such properties are average response time, number of successful invocations, etc. Current approaches only provide limited or no support in the consideration of non-functional properties in the service selection.
- *The client-side service management:* The WSML can also encapsulate the implementation of different client-side management concerns. Examples of such concerns are caching, billing, accounting, security, transaction, etc. Ideally these concerns need to be plugged-in and out at run-time according to the application requirements. Moreover, the consideration of unanticipated properties in the service selection implies the need for monitoring them by controlling how the services behave over time. For instance, if the application requires the fastest service, the performance of the involved services needs to be monitored. It is essential to insert and remove this monitoring functionality on demand, as the desired properties can dynamically change.

Dealing with these issues in the WSML weakens the link between the client application and the specific web services as all web service related code is taken out from the client application and placed in the WSML. As a result applications become more robust and adaptable to changes in the environment.

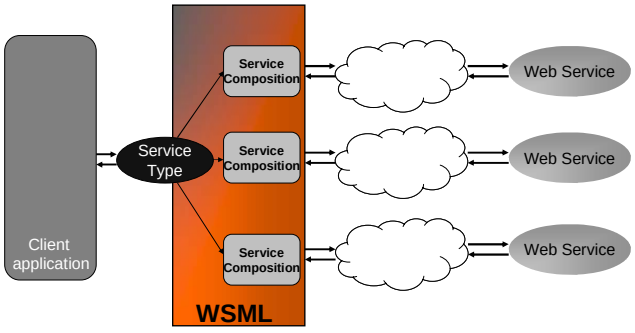

Figure 1. Role of the WSML

3 The need for dynamic AOP

Current approaches for the integration of web services are static and lead to unmanageable applications. To deal with the previously identified issues, management code has to be written manually and repeated for each service, resulting tangled with the core functionality and scattered all over the application. This becomes an obstacle for future maintenance. We observe that AOP is ideal to modularize these concerns. Moreover, in service-oriented applications a high flexibility is desired to be able to adapt and respond to changes in the environment, such us network problems or failure and unavailability of services, without having to stop the application. Support for dynamic aspects is needed to encapsulate these concerns as part of the WSML. As a consequence the client application becomes independent of specific services and more adaptable and robust.

In the WSML, three kinds of dynamic aspects are identified.

3.1 Redirection aspects

To decouple the client from specific services, the notion of *Service Type* is introduced in the WSML: a generic specification of the required functionality without references to specific web services. Then, concrete services can be registered to provide the functionality specified in a service type. Next, client applications can make a service type request and the WSML translates these generic requests to concrete web services invocations. In this mechanism we identify *redirection aspects*, which define the logic of intercepting client application requests and replacing them with concrete web service invocations. As such they encapsulate all *communication details* for a specific service or service composition. In addition, redirection aspects need to be dynamically plugged-in and out to reflect the volatility of the service environment.

3.2 Selection policy aspects

By default, the WSML selects web services and service compositions based on their availability. If a web service or a service composition is up and running and is reachable over the network it can be selected and invoked by the WSML. Ideally this default behaviour should be changeable at runtime to incorporate business driven requirements. Selection policies should be considered to take into account non-functional requirements. This service selection code typically crosscuts the client application. Thus we observe the need to encapsulate it in *selection aspects*. A selection policy aspect is a generic reusable aspect modularizing one selection policy. As selection criteria are based on business knowledge, they tend to change faster than the core application. Therefore, the selection aspects need to be dynamically pluggable.

3.3 Management and monitoring aspects

To decouple and cleanly modularize client-side management code in the WSML, *management aspects* are identified. For instance, all code that deals with client-side billing of a service resides in a billing aspect. An important management concern for web services is monitoring. Different service properties need to be watched constantly (e.g. response time, speed, latent time, price, etc) and then monitoring points need to be introduced dynamically in different places. To this end, monitoring aspects are identified.

4 Implementation of the WSML

In the previous section we identify different aspects in the selection, integration and management of web services that are part of the WSML. As the dynamic requirement is of essential importance, the AOP technology used for the implementation of the identified aspects should

provide support for the dynamic pluggability of aspects. A fully implemented version of the WSML was realised in the context of the MOSAIC[1] project and is available at [10]. This prototype is implemented in Java and uses JAsCo for implementing the identified aspects since its features are ideal to achieve the desired flexibility. Next section gives a brief introduction to JAsCo.

4.1 JAsCo

In [8, 9] the JAsCo aspect-oriented programming language that is tailored for the component based context is presented. JAsCo builds on top of Java and introduces two additional entities: aspect beans and connectors. An aspect bean is an extended version of a regular Java bean and allows describing crosscutting behaviour by means of a special kind of inner class, called a hook. Aspect beans are specified independently of concrete component types and API's, making them highly reusable. A connector on the other hand, is used for deploying one or more aspect beans within a concrete component context. In addition, connectors are able to specify explicit precedence and combination strategies in order to manage the cooperation among several aspects that are applicable onto the same join point. In addition, the JAsCo technology provides an extensive run-time infrastructure. Using this infrastructure, aspects remain first-class entities at run-time and dynamic aspect addition and removal becomes possible.

4.2 Realizing dynamic aspects using JAsCo

The current WSML implementation includes support for automatic aspect generation by means of creating JAsCo connectors and aspect beans at runtime, making it possible to integrate new services not anticipated at deployment time as well as new selection criteria and management concerns. Communication with the client application and the Web Services is done using Web Service technology. Therefore, the clients and services can run on different platforms.

Figure 2 illustrates the overall architecture of the WSML using JAsCo aspect beans and connectors. To implement the redirection mechanism based on service types and redirection aspects, JAsCo aspect beans are used to encapsulate the orchestration for a single service or service composition. JAsCo connectors are defined to link the aspect beans with the appropriate service type. The WSML is responsible for the creation and management of these redirection aspects and connectors. By creating a new connector and redirection aspect bean at runtime the new service or service composition can be integrated in the client application in a transparent way. The current version of the WSML supports full automatic generation of the connectors and provides tool support for the creation of the aspects. The proposed mechanism also enables hot swapping between services. If the response time of a service is too slow or the service becomes unavailable the selection module can "hot-swap" to another service by simply activating its connector and deactivating the previous one.

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

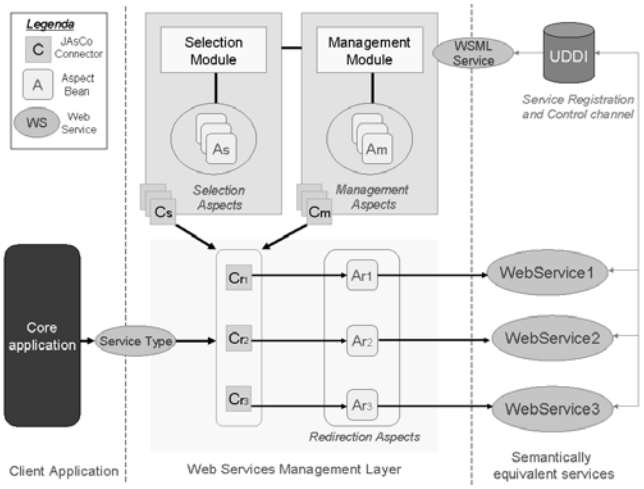

Figure 2. Detailed Architecture of the WSML

Regarding service selection, for each service policy, one connector and one aspect bean instance is created. Because of the reusability of JAsCo aspect beans, a selection policy aspect can be instantiated and initialised with different parameters at runtime by creating new connectors. As such a wide variety of selection policies can be specified and reinforced at runtime. Selection policies can be specified for a whole service type or for each request of a service type individually to realise a more fine-grained selection.

For the management aspects, a similar approach is followed. Each concern is modularized as an aspect and can be instantiated in different connectors. For example, a monitoring aspect can be instantiated at different monitoring points. Also, JAsCo combination strategies are used to control the order and exclusion of management concerns. For instance, if a caching aspect bean is instantiated to return a cached value instead of invoking an actual web service, a billing aspect bean encapsulating a billing strategy, does not need to be applied. For code examples we refer to [7].

The implementation of the WSML was very challenging as many aspects have to be dynamically defined and managed while guaranteeing a high performance. These characteristics influenced the selection of the AOP technology employed, since it had to provide features to dynamically and efficiently add and remove aspects. To illustrate the complexity in the management of aspects we made an approximation of the number of redirection aspects in a typical scenario of the WMSL. For each web service or service composition, (n+1) connectors and (n+1) aspect bean instances are created, where n is the number of requests of a service type. Assuming a typical service type is composed of 5 requests, and 10 web services and service compositions are registered in the WSML for this service type, there will be 60 connectors and aspect bean instances. Furthermore, the WSML is client application independent and can support multiple service types at once for different clients. This results in even higher numbers of connectors and aspect bean instances, making performance an important requirement of JAsCo. As JAsCo includes a state-of-the-art **ju**s**t**-in-**t**ime combined **a**spect compiler (Jutta) it is one the most optimal dynamic AOP implementations available.

5 Related Work

Other approaches also focus on the client-side service management: The Web Service Description Framework (WSDF) [11] incorporates ideas from the Semantic Web community [12, 13] suggesting an ontological approach for the invocation of services. The Web Services Mediator (WSM) [14] also identifies the need for a mediation layer to achieve dynamic integration of services. However, as far as we know there is no implementation available. The Web Services

Invocation Framework (WSIF) [15] supports a Java API for invoking web services irrespective of how and where the services are provided. WSIF mostly focuses on making the client unaware of service migrations and change of protocols.

The idea of applying AOP concepts to decouple web services concerns is quite innovative and thus not many approaches have been developed focusing on this field. Nevertheless, Martin et al. [16] have recently identified the suitability of AOSD to modularize the heterogeneous concerns involved in web services. However, they focus on approaches such as AspectJ [17] and HyperJ [18, 19] which, on the contrary to JAsCo, only allow static aspect weaving.

6 Conclusions

In this paper we presented the WSML as an example web service management framework where dynamic AOP is needed to modularize different service management concerns.

The WSML introduces a *dynamic binding* between client applications and web services. This makes the client applications more adaptive as services can be easily added and removed. Also, applications become more robust as they do not depend on specific services anymore. This dynamic binding mechanism is based on redirection aspects. We are working on automating this process by adding support for the semantic web. Adding semantic documentation to both the service types and the web services would make possible to automatically perform compatibility checks and generate glue code.

The basic hot-swapping mechanism can be enriched by defining selection aspects that take into account specific application requirements based on non-functional properties of services. Furthermore, the WSML defines aspects to encapsulate several client-side service management concerns. We are working on the specification of a generic and reusable library of selection and management aspects that are easily instantiated and deployable at run-time. This library would facilitate the dynamic pluggability of service selection and management concerns.

References

[1] W3C, "Simple Object Access Protocol (SOAP) v1.2," *Whitepaper, W3C Technical Publications,* http://www.w3.org/TR/SOAP/

[2] W3C, "Web Service Description Language (WSDL) v1.2," *Whitepaper, W3C Technical Publications,* http://www.w3.org/TR/wsdl12/

[3] Uddi.org, "Universal Description, Discovery and Integration," *UDDI Executive Whitepaper*, November 2001.

[4] Szyperski, C. "Components and Web Services," *Beyond Objects column, Software Development*, August, Vol. 9, No. 8., 2001

[5] Malhotra, J. "Challenges in Developing Web Services-based e-Business Applications," *Whitepaper interKeel Inc.*, 2001

[6] Verheecke, B., Cibrán, M. A., Jonckers, V., "AOP for Dynamic Configuration and Management of Web services in Client-Applications", Proc. 2003 International Conference on Web Services - Europe (ICWS'03-Europe), Erfurt (Germany), September 2003 and to be published in the International Journal on Web Service Research (JWSR) 2004

[7] Cibrán M.A., Verheecke B., Jonckers, V., "Modularizing Client-Side Web Service Management Aspects," *Proc. 2nd Nordic Conference on Web Services* (NCWS'03), Växjö (Sweden), 2003

[8] Suvée, D., Vanderperren, W., "JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development," *Proc. 2nd Int. Conf. on Aspect-Oriented Software Development (AOSD 2003)*, USA, 2003

[9] Vanderperren, W., Suvée D., Wydaeghe B., Jonckers, V. (2003), "PacoSuite & JAsCo: A visual component composition environment with advanced aspect separation features," *Proc. 2nd Int. Conf. on Fundamental Approaches for Software Engineering (FASE 2003)*, Warshaw (Poland), April, 2003

[10] Verheecke, B., Cibrán, M. A., "Web Services Management Layer (WSML)," http://ssel.vub.ac.be/wsml/

[11] Eberhart, A., "Towards Universal Web Service Clients," *Proc. Euroweb*, UK, 2002

[12] McIlraith, S., Martin, D., "Bringing Semantics to Web Services," *IEEE Intelligent Systems*, vol. 18, no.1, Jan/Feb 2003, pp 90-93

[13] Hendler, J. (2003), "Ontology-Enabled Pervasive Computing Applications", IEEE Intelligent Systems, vol. 18, no. 5 Sept/Oct 2003, pp 68-72

[14] Chatterjee, S., "Developing Real World Web Services-based Applications," *The Java Boutique*, http://javaboutique. internet.com/articles/WSApplications/

[15] Apache, "The Web Services Invocation Framework (WSIF)," http://ws.apache.org/wsif/

[16] Arsanjani, A., Hailpern, B., Martin, J., Tarr, P., "Web Services Promises and Compromises," *ACM Queue*, March, Vol. 1 No. 1, 2003, http://www.acmqueue.org/

[17] Kiczales, G., Hilsdale, E., Hugunin, J., Kersen, M., Palm, J., Griswold, W.G., "An overview of AspectJ," *Proc. European Conference on Object-Oriented Programming*, Hungary, 2001

[18] Ossher, H., Tarr, P. "Using multidimensional separation of concerns to (re)shape evolving software," *Communications of the ACM*, Vol. 44 No. 10, October 2001, pp.43—50

[19] Tarr, P., Ossher, H., Harrison, W., Sutton, S.M., "N degrees of separation: Multi-dimensional separation of concerns," *Proc. Int. Conf. on Software Engineering* (ICSE 1999)

A Dynamic Join Point Model for Java Object Lifecycle

Matthew Webster
matthew_webster@uk.ibm.com
IBM Corporation, Hursley, UK

Abstract

This position paper proposes the development of a dynamic join point model for Java™ object destruction to complement the static join points used to determine object initialization. Such a model would facilitate the implementation of efficient caching and pooling schemes in Java programs by allowing them to determine object lifetime.

This paper is based on my experience of GC in the Java Virtual Machine and my desire to apply Aspect Oriented Programming techniques to the implementation of system-wide caching and pooling.

1 Introduction

A cross-cutting concern frequently cited as an ideal candidate for implementation using AOP is caching [1]. Another is object pooling. However while many AO approaches like AspectJ [2] support join points for object initialization, equivalent join points are not available to determine when objects are eligible for garbage collection. This is because static join point models typically rely on source code or byte-code and there is no Java equivalent of the C++ destructor. Instead the garbage collector determines when objects are no longer reachable and recovers them.

A dynamic join point model would allow the interception of object lifecycle *before* destruction. Logic attached to the join point could determine whether the object can be reused and returned to the pool or left to be recovered by the garbage collector.

1.1 Object Caching and Pooling

Caching schemes can only be partially factored out using AOP. Object creation can be intercepted and either a new object or an existing one from the cache returned. However the maintenance of the cache must rely on other mechanisms. A cache can be cleared manually at a specific application phase boundary or automatically using Java reference objects [3]. The former approach is inflexible while the latter results in an empty cache after each garbage collection cycle.

The use of AOP to implement object pooling relies on an existing programming model and adherence to that model by the application. Such a model will typically include an “open” method called after object creation and a “close” method called when the object is no longer needed. One example is the JDBC java.sql.Connection class. An instance is obtained explicitly through a factory method on the java.sql.Driver class and a close method called when it is no longer required. Interception of both these methods could be achieved using an aspect and a pooling mechanism used to either restrict the number of open connections or reduce the overhead of creating a new instance each time. A Connection object is an expensive resource typically using a TCP/IP socket. However it is when applications do not adhere to the programming model that problems occur in large systems. For example if the “close” method is not called the Connection will not be returned to the pool but remain open until the next GC cycle.

1.2 Limitations of the Java Class Libraries

The Java language and accompanying class libraries offer several ways to interact with the garbage collector. Reference objects provide a mechanism for a program to be notified when an object is no longer reachable while the use of finalizers allows resources associated with an object to be released before it is garbage collected. Both mechanisms have drawbacks when used to implement either caching or pooling systems. In particular notification only takes place when a complete garbage collection occurs which may be infrequent for large heaps.

An instance of the java.lang.ref.Reference class can only be used to determine when its referent has already been collected. It is used in cache implementations to determine when an entry should be removed. It is of no use for implementing object pools.

To use a finalizer a class must override the "finalize" method which will be invoke on objects which are eligible for collection. Unlike when using a Reference the object has not been collected at this point and can in fact be made reachable again. However the JVM specification requires that the "finalize" method is only called once. If finalizers are to be used to implement an object pool a new finalizer must be created for each use of an object which is not very efficient.

2 Implementation

A dynamic join point model for object lifecycle requires regular notification from the garbage collector on the reachabitly of objects. Such events must occur more regularly than full collection. To make this process efficient pointcuts must provides hints as to which objects are being intercepted.

Escape analysis [4], applied either statically or at run-time by a JIT compiler, can be used to determine the scope of an object and optimize its allocation. Such a mechanism could also be used to notify the join point model when an object is no longer reachable. Such analysis may not always be efficient or guarantee to give a certain answer so would be best used to supplement other mechanisms.

2.1 Challenges

The implementation of a join point model for object lifecycle will requires a modified Java VM. The Jikes RVM [5] provides an excellent platform for experimentation but longer term adoption would require incorporation into a commercial standard JVM.

Another challenge is path length. Object allocation in the Java VM has been highly optimized and so has garbage collection. An AO implementation of a pooling mechanism must be extremely efficient or be reserved for objects whose construction is expensive or relies on non-Java resources such as TCP/IP sockets or operating system files.

2.2 Other Approaches

Traditional mainframe-based transaction processing environments such as the IBM Customer Information Control System (CICS) [6] present a particular challenge to the use of the Java language for application programs. On large systems a throughput of many thousands of transactions per second is not uncommon, resulting in a short application lifetime in relation to garbage collection pause times. To meet this challenge an implementation of a "serially reusable" [7] JVM was incorporated into the IBM SDK for z/OS™ Java 2 Technology Edition, Version 1.3 and Version 1.4. Objects are placed at different locations in a partitioned heap according to their expected lifetime which is determined by class and class loader. The user is then notified if a particular object has an actual lifetime which meets this expectation and is given the opportunity to modify the configuration. While this is a powerful mechanism for applications written to a restricted programming model such as J2EE it does not allow instances of the same class to have widely varying lifetimes.

3 Summary

My background is Java Virtual Machine implementation and the development of advanced garbage collection models. My current activities are the development of AOSD and its application to complex middleware. I believe that the best performance and flexibility of such systems will be achieved by implementing cross cutting concerns such as caching and pooling using AOP.

Acknowledgements

I would like to thank my colleagues in the performance team for listening to my ideas.

References

[1] Jonathan Davies, Nick Huismans, Rory Slaney, Sian Whiting, Robert Berry, Matthew Webster "An Aspect Oriented Performance Analysis Environment", http://aosd.net/archive/2003/program/davies.pdf

[2] AspectJ, http://www.eclipse.org/aspectj/

[3] Java 2 SDK, Standard Edition, Documentation, http://java.sun.com/j2se/1.4.2/docs/guide/refobs/index.html

[4] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, Sam Midkiff, "Escape Analysis in Java", http://www.research.ibm.com/jalapeno/publication.html#oopsla99_escape

[5] Jikes Research Virtual Machine, http://www-124.ibm.com/developerworks/oss/jikesrvm/

[6] CICS, http://www-306.ibm.com/software/htp/cics/

[7] Sam Borman, Susan Paice, Matthew Webster, Martin Trotter, Rick McGuire, Alan Stevens, Beth Hutchison, Robert Berry, "A Serially Reusable Java VM", http://www-1.ibm.com/servers/eserver/zseries/software/java/pdf/29.3406.pdf

Towards an Efficient Aspect Precedence Model

Yang Yu
School of Computer Science
McGill University
yyu@cs.mcgill.ca

Jörg Kienzle
School of Computer Science
McGill University
Joerg.Kienzle@mcgill.ca

Abstract

In systems that make heavy use of aspect-oriented programming to encapsulate different application concerns, chances are high that several aspects add functionality to the same part of the program code, i.e. they apply to the same join point. In certain cases the order of execution of aspect functionality at a join point is crucial to achieve correct system behavior, for example, when there are dependencies between aspects, or when the functionality delivered by one aspect should override the functionality of the other one. This paper first illustrates the necessity of prioritizing aspects that apply to the same join point, and then examines the sequential aspect precedence model provided by AspectJ. The paper then explores a more general aspect precedence model, where aspects can potentially execute concurrently. The model is illustrated by means of an example involving authentication, authorization and logging. Finally, a theoretical performance comparison between the two precedence models is presented.

1 Introduction

Separation of concerns is a fundamental principle of software engineering that in its most general form refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal, task, or purpose. The benefits of a successful modularization of concerns during the implementation phase are obvious: simpler code structure resulting in improved readability of program code, program code that is easier to customize and adapt to new situations, increased possibilities for reuse.

When following an object-oriented approach to software development, a problem is decomposed into objects, each of them providing a well-defined part of the main functionality of the system. As a result, secondary functionality, e.g. distribution support, is often poorly encapsulated. This phenomenon is known as the "tyranny of the dominant decomposition" [1].

Aspect-Oriented Programming [2] provides a means for addressing this problem. It allows programmer to modularize such secondary functionality, also called cross-cutting concerns, in so-called aspects, and to describe their relationships. Ultimately, the programmer relies on the underlying AOP environment to *weave* (or compose) the concerns together into a coherent program. This composition takes place at well-defined points during the execution of the application. These points are called join points. AOP languages are supposed to provide means to identify join points, specify behavior at join points, define units that group together join point specifications and behavior enhancements, and provide means for attaching such units to a program.

Most large software systems encompass a number of aspects, some of which might be very general, e.g. pooling, distribution, caching, authorization, fault tolerance, etc. As the number of aspects in a system increases, some aspects may apply to the same join point. In some cases, the order of execution of aspect functionality at a certain join point is crucial to achieve correct system behavior. For instance, caching must be applied before distribution, authentication must

occur before allowing access to sensitive data, etc. In short, some aspects are more important for the application, or depend on functionality offered by other aspects, or might want to override functionality offered by others. Aspect-oriented programming environments must provide means for specifying these precedence rules.

This paper introduces a general precedence model that makes it possible to determine an optimal aspect execution strategy for aspects that apply to the same join point. Section 2 shows how one might implement general aspects such as authentication, authorization and logging in AspectJ, and how a programmer can specify the precedence of these aspects when they apply to the same join point. This example is followed by a complete presentation of the sequential precedence model of AspectJ in Section 3. Section 4 identifies the properties of aspects that are of interest when trying to determine a valid aspect execution order. Based on these properties, Section 5 then defines a more general, concurrent precedence model. Section 6 presents a theoretical performance comparison between the two models, and Section 7 discusses implementation issues.

2 AspectJ Examples

We first investigated the precedence model of AspectJ [3], an aspect-oriented programming environment for the Java language. In AspectJ, join points are certain well-defined points in the execution flow of a Java program. These include method and constructor calls or executions, field accesses, object and class initialization, and others. *Pointcut* designators allow a programmer to select a certain set of join points, which can further be composed with boolean operations to build up other pointcuts. It is also possible to use wild cards when specifying, for instance, a method signature.

The following code, for instance, defines a pointcut named `CallToAccount` that designates any call to a public method of the `Account` class:

```
pointcut CallToAccount () : call (public * Account.*(..));
```

To define the behavior at a join point, AspectJ uses the notion of *advice*. An advice contains code fragments that execute *before*, *after* or *around* a given join point. *Around* means that the code in the advice executes instead of the code of the join point it applies to. However, at any point in time the *around* advice code can execute a special *proceed()* statement that will pass control to the code that was intended to execute at the join point in the first place. This is, however, not mandatory. Hence, *around* advice allow a programmer to skip or replace the code of a join point entirely. The concepts of *before*, *after* and *around* advice are illustrated in Figure 1.

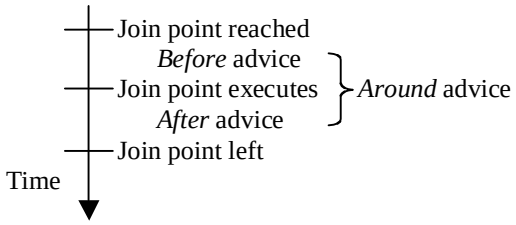

Figure 1. AspectJ Advice Kinds

Finally, AspectJ defines the notion of *aspects*. Aspects, very much like a class, group together methods, fields, constructors, initializers, but also named pointcuts and advice. These units are intended to be used for implementing a crosscutting concern.

A typical banking system, for example, must be able to handle, besides its core functionality, the concerns of authentication and authorization. Prior to performing any operations on account objects, a client must have been authenticated and authorized. It is of course important that the

authentication is done before the authorization. In AspectJ, the authentication and authorization can be encapsulated in two aspects as shown in Figure 2 and Figure 3. The code is inspired by examples from [5]:

```
public aspect AuthenticationAspect  {
  public static Subject
    authenticatedSubject = null;
  public pointcut ①
    authenticationOperations():
    execution(public * Account.*(..))
    || execution(public *
      InterAccountTransferSystem.*(..));

  before(): authenticationOperations() { ②
    if(authenticatedSubject != null) {
      return;
    }
    try {
      authenticate();
    } catch (LoginException e)  {
      throw new
        AuthenticationException(e);
    }
  }

  private void authenticate() throws
      LoginException {
    LoginContext lc = new
      LoginContext("Sample",
        new TextCallbackHandler());
     lc.login();
     authenticatedSubject =
       lc.getSubject();
   }

  public static class
    AuthenticationException
    extends RuntimeException {
      public AuthenticationException
        (Exception e) {
        super(e);
      }
    }
  }
}
```

Figure 7. Authentication Aspect

Both aspects declare a pointcut (see ① in Fig. 1 and Fig. 2) that designates all method execution points in the `Account` or `InterAccountTransferSystem` classes. In the example, both aspects also declare *before* advice that are triggered by the same pointcut, i.e. their code will execute prior to any code inside the `Account` or `InterAccountTransferSystem` classes. The authorization aspect also defines an *around advice* that lets it control if the actual join point, i.e. the method execution, should be processed or not.

```
public aspect AuthorizationAspect  {
  public pointcut ①
    authorizationOperations():
    execution(public * Account.*(..))
    || execution(public *
      InterAccountTransferSystem.*(..));

  before(): authorizationOperations() {
    if (AuthenticationAspect.
        authenticatedSubject  != null) {
      -- ask for permission based on the
      -- method name
      Permission permission = new
        BankingPermission
     (joinPointStaticPart.getSignature().
      getName());②
      AccessController.checkPermission
        (permission);
    }
  }

  Object around():
     authorizationOperations() &&
    !cflowbelow(authorizationOperations())
    { if (AuthenticationAspect.  ②
     authenticatedSubject  != null)  {
       try {
         return Subject.doAsPrivileged(
           AuthenticationAspect.
authenticatedSubject, new PrivilegedExceptionAction() {
  public Object run() throws
    Exception { return proceed(); }
  }, null);
       } catch
         (PrivilegedActionException e) {
         throw new
           AuthorizationtionException
             (e.getException);
       }
     }
   }
   public static class
     AuthorizationtionException
       extends RuntimeException {
       public Authorizationtion
         (Exception e){
         super(e);
      }
   }
}
```

Figure 2. Authorization Aspect

Obviously the two aspects are not independent. The authorization aspect depends on functionality provided by the authentication aspect. In order to be able to handle proper authorization, the authorization aspect must be able to obtain the authenticated subject, which is produced by the authentication aspect (see ② in Fig. 2). The authentication aspect somehow is more important, or provides low-level functionality, which the authorization aspect depends on.

This example illustrates that, in the case where multiple advice apply to the same join point, the notion of aspect precedence may be of importance.

In addition to authentication and authorization, we might want to log all calls made to `Account` or `InterAccount-TransferSystem` classes. This very general functionality can easily be implemented in AspectJ as shown in Figure 4. The LoggingAspect declares a *before* advice that prints out logging information before the actual method executes.

```
public aspect LoggingAspect  {
  public pointcut ①
    loggingOperations():
    execution(public * Account.*(..))
    || execution(public *
      InterAccountTransferSystem.*(..));

  private Logger logger =
    Logger.getLogger("trace");

  before(): loggingOperations() {
    Signature sig =
    thisJoinPointStaticPart.getSignature();
    if(logger.isLoggable(Level.INFO)) {
      logger.logp(Level.INFO,
        sig.getDeclaringType().getName(),
        sig.getName(),
        "Performing operation " +
        sig.getName() +
        "on the object " +
        sig.getDeclaringType().getName());
    }
  }
}
```

Figure 4. Logging Aspect

3 AspectJ Precedence Model

In AspectJ, advice attached to the same join point are executed sequentially. The order of execution depends on the kind of advice, i.e. *before*, *after* or *around*, and the aspect precedence. The precedence is derived from explicit precedence declarations made by the programmer, or else determined based on the default precedence rules. These default rules are described below.

In the following description, AS_i and AS_j are two aspects, AD_i and AD_j are two advice of the same kind (e.g. before advice), and finally $P(x)$ is the priority of aspect x.

3.1 Aspect Precedence Rules

- If AS_i is derived from AS_j, then AS_i has precedence over AS_j .
- If there is no inheritance relationship between AS_i and AS_j, then:
  - If there is an aspect precedence declaration between AS_i and AS_j, then the aspect who occurs first in the precedence declaration list has the precedence over the other aspect.
  - If there is no aspect precedence declaration between AS_i and AS_j, then the aspect whose name has a *higher string value* has the precedence over the other aspect.

3.2 Advice Execution Ordering Rules

If two advice AD_i and AD_j are in two different aspects, then their execution order is determined based on the rules stated in Subsection 3.1. If $P(AS_i) > P(AS_j)$ then AD_i has precedence over AD_j, meaning that if AD_i is a before advice or around advice it will execute before AD_j, if it is an after advice it will run after AD_j. But what happens if the advice are both declared in the same aspect?

In this case, precedence depends on the set of join points each advice applies to. Let pc_i and pc_j be pointcuts attached to AD_i, respectively AD_j.

- If pc_j is a subset of pc_i, then AD_i has predence over AD_j.
- If pc_i is identical to pc_j, or if pc_i and pc_j are different but not subsets of each other, then the precedence is determined based on the textual appearance of AD_i and AD_j in the aspect source code. If AD_i is placed textually before AD_j, then AD_i has precedence over AD_j.

3.3 AspectJ Precedence Declaration

According to the precedence rules of AspectJ, the authorization aspect shown in Figure 3 has precedence over the authentication aspect shown in Figure 2, since the authorization aspect has a name with a higher string value ("AuthorizationAspect" > "AuthenticationAspect"). In order to make the authentication aspect have precedence over the authorization aspect, an explicit precedence declaration has to be added to the program:

```
public aspect AuthOrder {
  declare precedence:
    AuthenticationAspect,
    AuthorizationAspect;
}
```

Figure 8. AspectJ Precedence Declaration

With this declaration, the before advice in the authentication aspect will execute before the before advice in the authorization aspect.

3.4 Sequential Precedence Model

Figure 6 describes the complete precedence model of AspectJ. AS_1, AS_2, …, AS_n are aspects applying to the same join point. Their precedence is descending, i.e. $P(AS_1) > P(AS_2) > \ldots > P(AS_n)$. Each aspect declares a set of *before* advice, AD_b, a set of *around* advice, AD_r, and a set of *after* advice, AD_a. In the case where there are no explicit precedence declarations, the execution sequence of advice is as follows:

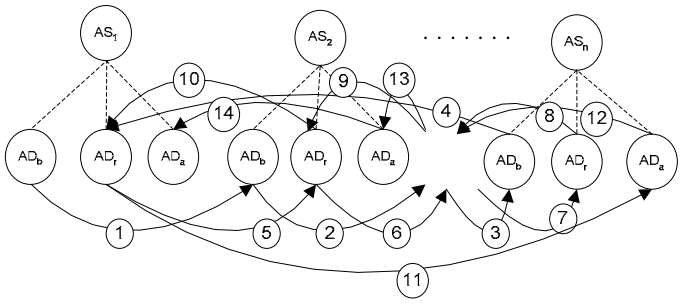

Figure 6. Sequential Precedence Model of AspectJ

First all the *before* advice execute, i.e. AD_{b1} .. AD_{bn} (1–3), then the *around* advice, i.e. AD_{r1} .. AD_{rn}. Obviously, if one of the around advice does not call *proceed()*, the advice that follow will not execute. Think of the *around* advice execution like nested procedure calls. If all around advice execute a *proceed()* statement, then the actual join point will be reached and executed. In Figure 6 this would result in executing 5-7, and then the actual join point. Finally, execution returns through all the around advice to execute any remaining code in opposite order, i.e. AD_{rn} .. AD_{r1} (8-10). Finally, the after advice execute in inverse precedence order, i.e. AD_{an} .. AD_{a1} (11-14).

The case where advice AS_i is explicitly declared to have precedence over AS_j is illustrated in Figure 7.

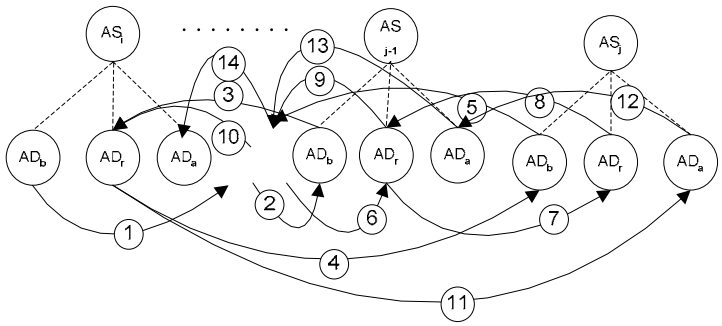

Figure 7. The Sequential Precedence Model of AspectJ with Precedence Declared Aspects

First all before advice of AS_i … AS_{j-1} execute (1-2), then the around advice of aspect AS_i (3), then the before advice of AS_j (4), and then all around advice of AS_{i+1}… AS_j execute (5-7), and then the join point is reached. After execution of the join point, the remaining code of the around advice are executed in reverse order (8-10), the last one being the on of AS_i. Finally, all after advices of AS_j … AS_i execute (11-14), the last one being the after advice of AS_i (14).

3.5 Discussion

The model imposes strict nesting at the aspect level, similar to what programming languages do for nested method invocations. Given two aspects AS_i and AS_j, each aspect declaring a before and an after advice, it is, for instance, impossible to achieve the execution order: $AS_{i\ before}$, $AS_{j\ before}$, joinpoint, $AS_{i\ after}$, $AS_{j\ after}$. It turns out, however, that such an ordering is rarely needed. It can anyhow be achieved, for instance, by putting the before advice of aspect AS_j in a new aspect AS_{j1} and the after advice of aspect AS_j in a new aspect AS_{j2}. The precedence declaration $AS_{j2} > AS_i > AS_{j1}$ then results in the desired ordering.

The AspectJ precedence model is also a little confusing because of the way it handles different kinds of advice. Conceptually, putting code in a *before* advice, or in the first part of an *around* advice should be equivalent![1] This drawback can be overcome, though, by only using, for example, *around* advice.

In certain cases, however, the AspectJ precedence model might be too restrictive because it enforces serial execution of advice. Consider a very general logging aspect that logs method invocations of account objects for debugging purposes. It might not be important *when* the logging happens, i.e. before or after the method call. It could even happen *while* the actual call is executed. On a hyperthreaded processor or in a multiprocessor system executing the logging advice concurrently will result in a performance increase. If logging involves lengthy I/O operations such as writing to a file, accessing a database or making a remote method invocation, then performance will increase even in single processor environment!

[1] This is why in Section 4 we have chosen to distinguish between before code and after code only.

In order to make concurrent execution possible, a more general precedence model is needed that allows for a more efficient and flexible scheduling of advice execution at join points.

4 Properties of Advice

Before we proceed to define a general precedence model, we will concentrate on the essential properties of advice that are relevant to determine their precedence.

To simplify our model, we make no distinction between what AspectJ calls an advice and an aspect, i.e. the unit that adds behavior (the code) and the encapsulation mechanism (the module)[1]. Also, we only consider one kind of advice, which can define two code sections, one that executes before, and one that runs after a certain join point execution. Either code section can be empty. In order to obtain the behavior of an AspectJ *around* advice, the advice code can specify if the actual join point (or lower priority advice) should be executed between the before and after code or not.

```
Advice:
  [before code]
  [join point execution]
  [after code]
```

Our flexible advice execution scheduling policy will determine execution ranges, i.e. earliest start and latest end time, for both *before code* and *after code*.

In order to do this, the programmer has to specify for every start and end time if it is *constrained* or *unconstrained*.

Unconstrained before starting time means that it does not matter when the before code starts executing. Constrained before starting time means that the before code can start execution at earliest once the actual join point has been reached. Obviously, the before start time has to be constrained if the advice code depends on runtime information that is available only when the join point is reached.

Unconstrained before code ending time means that the before code does not have to finish execution before the actual join point is reached. Constrained before code ending time implies that the before code must finish execution at latest before the actual join point is executed.

Unconstrained after code starting time means that it does not matter when the before code starts executing. Constrained after code starting time implies that the advice code can start executing at earliest after the actual join point has executed.

Unconstrained after code ending time means that the after code does not have to finish execution before the normal program execution continues after the join point. Constrained after code ending time implies that the after code must finish execution at latest before the normal program execution continues after the join point.

In addition to these individual constraints, the programmer can specify priorities between advice, for instance, "$A_i > A_j$", meaning that A_i has higher precedence than A_j. This is very similar to the way AspectJ handles precedence declarations. Such a declaration is necessary in cases where an aspect depends on the functionality of some other aspect, or cases when an aspect must decide about the execution of some other aspect or the actual join point.

Such a precedence declaration constrains the start and end times even further. For instance, declaring that an advice A_i takes precedence over an advice A_j results in constraining the before code end time of A_i to be equal or earlier than the before code start time of A_j. Also, it results in constraining the after code end time of A_j to be smaller or equal to the after code start time of A_i. In a sense, a precedence declaration between two advice creates a serial dependency.

[1] In AspectJ you can get the same effect by putting every advice in a separate aspect.

Advice that have either unconstrained start times or end times care called *flexible advice*, since their scheduling requirements are less stringent. Advice with unconstrained starting time are called *openstarted*; advice with unconstrained ending time are called *openended*.

5 A Concurrent Precedence Model

Based on the start and end time declarations, and on the different precedence declarations among different advice, an optimal advice execution schedule can be determined. The idea is to construct a dependency graph similar to the ones used in project planning software.

Every advice is represented as two nodes in the graph, one for the before code, one for the after code. In addition, there are three special nodes: the "join point reached" node *jr*, the join point execution node *je*, and the "join point left" node *jl*. Every precedence declaration "$A_i > A_j$" is reflected as a dependency link from the before code node A_{bj} to A_{bi}, and one from the after code node A_{aj} to A_{ai}. All before advice nodes with constrained before start times are linked from the *jr* node. All after advice nodes with constrained after end times are linked to the *jl* node.

Now, starting from all source nodes (i.e. all openstarted before and after advice and the *jr* node), we can determine all start times of successor nodes, and so on, until all the start times have been filled in. Similarly, we can determine all end times by starting from all sink nodes, i.e. the *je* node and all openended before and after advice, and going in the opposite direction.

Consider the AspectJ example of Section 2, i.e. authentication, authorization and logging. Authentication and logging only have before code, authorization has before and after code. We do not care when authentication actually happens, and it does not need any run-time information from the join point context (see ② in Figure 2). Therefore, the before start time can be set to *unconstrained*. The authorization aspect on the other hand needs run-time information (see ② in Figure 3), since authorization depends on the invoked method name. It's before start time is therefore constrained to be after the actual join point has been reached. The logging aspect writes a log message to the log when the join point is reached, and before the actual method executes. The before start time is therefore constrained. However, if we don't require that the logging is completed before the method starts executing we can set the before end time to unconstrained!

The authorization aspect depends on the authentication aspect, so we have to specify "$A_{authentication} > A_{authorization}$". The logging aspect is independent of the other aspects, as long as we do not want to log only authorized calls. The resulting precedence graph is shown in Figure 8.

The figure illustrates that we could take advantage of concurrent advice execution in this example by executing the logging before code in parallel with the authorization before code. Moreover, no other code depends on the logging code, so even if it executes slowly, the rest of the program execution is not affected. Once the authorization before code has executed, the program can safely continue and execute the join point itself.

It also becomes apparent that we can perform authentication even before we reach the join point. On one hand this might seem a strange thing to do, because the actual join point might never be reached, and therefore authentication might not be needed at all. On the other hand, there might be a time during the execution of the application where there is spare processing power. In such a case, authentication can be performed in advance, without any additional cost. If the join point is reached in the future, advice execution will be a lot more efficient.

In real-time systems, where usually the worst-case execution time of code is known, it is even possible to calculate the exact time span in which every advice should execute. Moreover, it is possible to determine the critical path of advice execution, and finally to obtain the worst-case execution time (WCET) of all advice and the join point by summing all the worst-case execution times of all advice lying on the critical path.

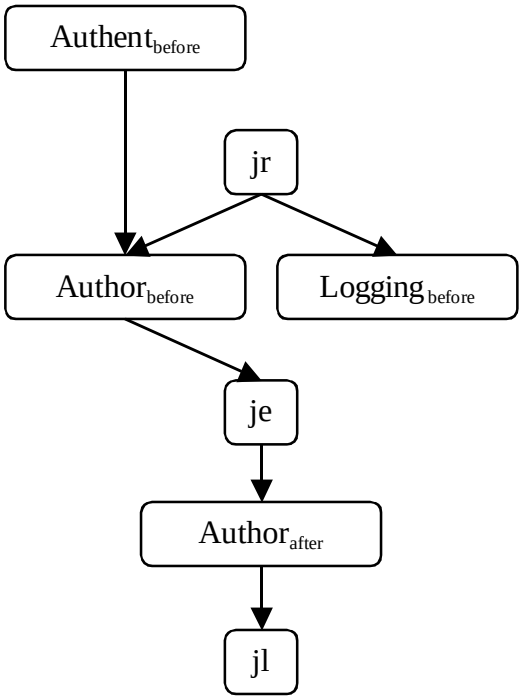

Figure 8. Advice Precedence Graph

If worst-case execution times of all advice code are known, then the precedence graph can also be shown in form of a PERT graph or Gantt chart. The following figure shows the Gantt chart for our example:

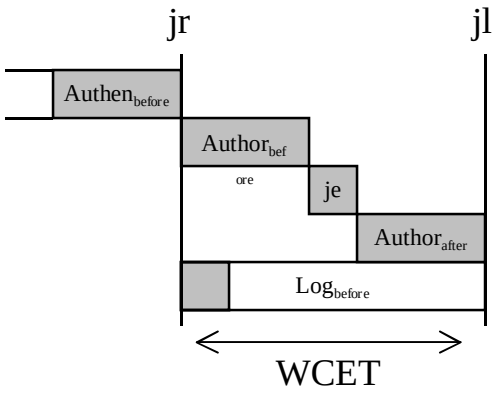

Figure 9. Gantt Chart Illustrating Advice and Join Point Worst-Case Execution Time

6 Performance Comparison

The performance comparison between the concurrent and the sequential model is based on Pre- or Post- Join Point Time (PJPT), which refers to the time quantum of advice execution between *jr* and *je* nodes for Pre-Join Point Time, or between *je* and *jl* nodes for Post-Join Point Time. Suppose there are *n* advices (AD_1, AD_2, …, AD_n) attached to a join point. A general advice precedence graph for these advice is depicted in Figure 10. $PJPT_s(AD_1, AD_2, \ldots, AD_n)$ and $PJPT_c(AD_1, AD_2, \ldots, AD_n)$ denote the PJPT times for the sequential, respectively for the concurrent model. $T(x)$ is the execution time of advice *x*.

For the sequential precedence model, the PJPT time can be calculated as follows:

$PreJPT_s(AD_1, AD_2, \ldots AD_n) = T(AD_{b1}) + T(AD_{b2}) + \ldots + T(AD_{bn})$
$PostJPT_s(AD_1, AD_2, \ldots AD_n) = T(AD_{a1}) + T(AD_{a2}) + \ldots + T(AD_{an})$

To calculate PJPT for the concurrent precedence model, we have to consider the different paths connecting the individual advice. We therefore need to define the notion $T_{path}(j,k)$, which is the worst-case execution time for the set of advice defined by all nodes that lie on all paths connecting node *j* and node *k*, not including the execution time of node *k*. If *Linked(k)* denotes the set of nodes linked to the node *k*, $T_{path}(j,k)$ can be defined as:

$T_{path}(j, k) =$
if $j = k \Rightarrow 0$,
if $j \neq k \Rightarrow \max_{i \in Linked(k)}(T_{path}(j,i)) + T(i))$

Given these definitions, the PJPT time for the concurrent precedence model is defined by:

$PreJPT_c(AD_1, AD_2, \ldots, AD_n) = T_{path}(jr, je)$
$PostJPT_c(AD_1, AD_2, \ldots, AD_n) = T_{path}(je, jl)$

In the best case there are only openstarted or openended advice attached to a join point, and hence the Pre / PostJPT for the concurrent model is zero (since all code can be executed concurrently to the join point execution), while the PJPT in the sequential model is equal to the sum of the execution time of all advice. In the worst case there is only one path from node *jr* to *je* or from *je* to *jl*, in which case the PJPT time is the same for both precedence models.

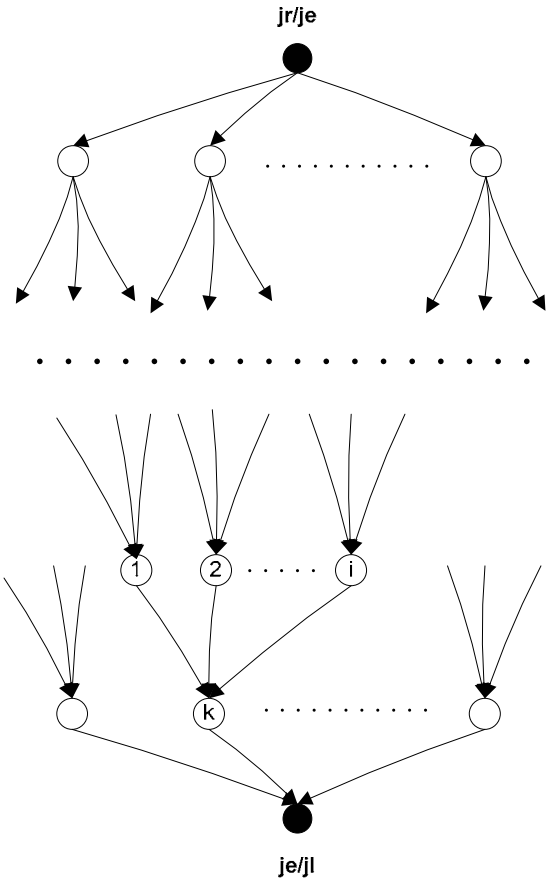

Figure 10. Advice Precedence Graph for calculating PJPT

7 Discussion

The proposed general aspect precedence model can result in an important performance gain when executing multiple advice attached to the same join point. This performance gain, however, is up to now only theoretical. It still needs to be shown that the model can be implemented efficiently.

There are several implementation issues that need to be solved. Thread creation is a time and memory consuming operation. If for every advice that executes concurrently a new thread has to be created, then the performance gain might be reduced, or worse, the resulting execution might even be slower than an equivalent sequential one. In order to overcome this problem, a single thread should be used for every sequential advice group. Also, unused threads should be kept in a thread pool for future use.

Another problem that needs to be solved are advice parameters and execution contexts. An advice code might make use of join point parameters, or might want to obtain run-time information from the virtual machine, for example, by using reflection. A means must be found to allow all threads that execute advice concurrently to share this information.

8 Conclusion

This paper investigated the problems that occur when several aspects or advice apply to the same join point. When these advice are related, it is often necessary to impose an ordering on their execution in order to achieve correct behavior. Existing aspect-oriented programming environments such as AspectJ allow a programmer to specify such an ordering by means of precedence declarations. However, even unrelated advice that do not need to be executed in a specific order will get an implicit precedence and are executed in some sequence.

This paper defined a more general aspect precedence model. It requires the programmer to specify start and end time constraints of advice, and to declare precedence relationships among advice that have semantic dependencies. Based on this information, an optimal advice execution schedule exploiting concurrency can be determined. In some cases it is even possible to start executing some advice code before the actual join point has been reached. Likewise, the model can help identify situations in which it is possible to continue program execution and advance to the next join point while there is still advice code related to the current join point running.

The presented model is particularly well suited for real-time systems, where the worst-case execution time of code is known. In this case, a detailed advice execution schedule can be established, and the overall worst-case execution time can easily be calculated.

References

[1] Tarr, P. L., et al.: “N Degrees of Separation: Multi-Dimensional Separation of Concerns”. In Proceedings of the 21st International Conference on Software Engineering (ICSE’1999), pp. 107-119, IEEE Computer Society Press / ACM Press, 1999.

[2] Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.: “Discussing Aspects of AOP”. Communications of the ACM 44 (10), pp. 33 – 38, October 2001.

[3] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersen, M.; Palm, J.; Griswold, W. G.: “An Overview of AspectJ”. In 15th European Conference on Object–Oriented Programming (ECOOP 2001), pp. 327 – 357, June 18–22, 2001, Budapest, Hungary, 2001.

[4] The AspectJ Team: Aspect User Guide. http://www.eclipse.org/aspectj/

[5] Ramnivas Laddad. *AspectJ In Action*. Manning Publications Co., 2003.