

Dynamic Weaving for Aspect-Oriented Programming*

Andrei Popovici , Thomas Gross, and Gustavo Alonso
Department of Computer Science
Swiss Federal Institut of Technology Zürich
CH-8092 Zürich, Switzerland
{popovici,trg,alonso}@inf.ethz.ch

ABSTRACT

When using Aspect Oriented Programming in the development of software components, a developer must understand the program units actually changed by weaving, how they behave, and possibly correct the aspects used. Support for rapid AOP prototyping and debugging is therefore crucial in such situations. Rapid prototyping is difficult with current aspect weaving tools because they do not support dynamic changes. This paper describes PROSE (*PROgrammable extenSions of sErVICES*), a platform based on Java which addresses dynamic AOP. Aspects are expressed in the same source language as the application (Java), and PROSE allows aspects to be woven, unwoven, or replaced at run-time.

1. INTRODUCTION

Important concerns of modern applications like transactions, security, distribution, or logging are not easily expressed in a modular way. *Aspect Oriented Programming* (AOP) [10] has emerged as a promising technique to overcome this problem. Aspects express functionality that cuts across the system, allowing the developer to design a system out of orthogonal concerns and to provide a single focus point for modifications.

There have been a considerable number of research efforts around AOP in the last few years. Tools like Hyper/J [18] allow programmers to divide and re-compose applications out of orthogonal concern spaces. AspectJ [21] is a language that provides a concise and secure way to express crosscutting functionality in Java programs. The composition filter approach [2] adds message-level filters to objects or collections of objects. Adaptive Plug-and-Play Components [12, 14] define generic behavior for a set of classes that can be personalized to different class models. Applications of AOP have emerged in areas like distribution and synchronization [13] or real-time [1]. Related techniques are used in the adaptation of the service layer in distributed system [19].

*Effort sponsored in part by the Swiss National Science Foundation NCCR MICS (Mobile Information and Communication Systems).

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

AOSD 2002, Enschede, The Netherlands

Copyright 2002 ACM 1-58113-469-X/02/0004...\$5.00.

Current implementations of AOP are based on compile-time or load-time modification of the application code. It has been observed that the whole spectrum of binding times (compiling, loading, run-time) is needed [4, 8, 17]. Dynamic weaving could increase the attractiveness of AOP for rapid prototyping and testing by avoiding the re-compilation, re-deployment and re-start of the application. Similarly, it would allow the use of AOP for adapting services in response to changes in the environment [8]. In such cases the adaptations may need to be performed as late as possible, ideally dynamically.

In this paper we present a platform, PROSE, that supports dynamic AOP. Aspects can be woven and unwoven at run-time, providing for a faster design-test cycle. Repeated weaving also helps developing aspects that must be added simultaneously to an application, specially if their interactions are difficult to understand at compile time. In PROSE, aspects are written in Java, and no separate tools (aspect weavers) are needed, because its implementation is based on the debugger interface of the virtual machine. By expressing aspects in the source language, PROSE allows the definition of customized AOP constructs. If an experimental weaving feature is needed, it can be incorporated into the platform. Recently, other authors have pleaded for more openness of the weaving process [20].

The rest of this paper is structured as follows: In Section 2 we present an aspect in PROSE. In Section 3 we comment on the implementation alternatives and the design of PROSE. We discuss in Section 4 the performance results we gathered with PROSE. We conclude the paper in Section 5.

2. AOP WITH PROSE

Aspect-orientation allows the composition of crosscutting concerns, a way to specify advice actions, and intuitive access to local environments. PROSE tries to match these requirements by providing a core subset of essential AOP features. The goal of PROSE is to allow the definition of aspects following existing AOP solutions, while avoiding the complexity of Meta Object Protocols.

2.1 An aspect in PROSE

Consider a hierarchy of classes that implements a base interface (`Printer`). An aspect defines functionality needed for access control and accounting. Weaving the aspect in the printer classes leads to the result in Figure 1.

The first method (`cancelJob`) contains the access control code (the gray rectangle) followed by the actual cancel operation. The `print`-method first contains the printing logic and then the accounting code (the hatched rectangle).

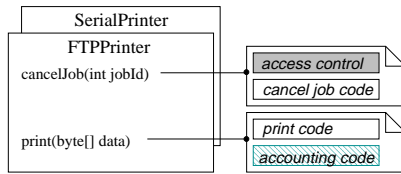


Figure 1: Code scattered in subclasses of Printer.

Both the access control code and the accounting code are present in all subclasses of `Printer` (e.g., `SerialPrinter`, `FTPPrinter`).

Figure 2 shows the PROSE aspect for weaving the access control code *before* the actual body of methods matching "cancel". The accounting code is to be woven *after* the body of all methods named `print` in all classes inheriting `Printer`. This PROSE example is a well-formed Java class.

```

1 class ExampleAspect extends Aspect {
2     Crosscut doAct1 = new FunctionalCrosscut() {
3         public void ANYMETHOD(ANY anyThis, REST rst) {
4             // access control code }
5         { setSpecializer(
6             (MethodS.named(".*cancel.*")).AND
7             (MethodS.BEFORE)); }
8     };
9     Crosscut doAccnt = new FunctionalCrosscut() {
10        public void print(ANY anyThis, byte[] b) {
11            // accounting code }
12        { setSpecializer(
13            (MethodS.AFTER).AND
14            (Classes.extending(Printer.class))); }
15    };
16 }

```

Figure 2: A PROSE aspect for weaving the access control and accounting functionality.

Aspects in PROSE must extend the class `Aspect` (line 1). A class inheriting `Aspect` may contain one or more crosscut objects¹ declared as instance variables (lines 2 and 9). Each crosscut defines a set of join-points in the original application and an advice to be executed at the join-points. The first crosscut object (`doAct1`) contains an advice method named `ANYMETHOD` (line 3) that defines the action to be taken upon entry of all "cancel" methods. The body of this method contains code (line 4) that can be taken from the old printer code in Figure 1. The special signature of the method on line 3 defines a *set* of locations where its body must be executed. The first parameter of the method stands for the type of the receiver of the call. This parameter has the type `ANY` and matches all object types. The second argument has the type `REST` and matches lists of parameters of arbitrary types. Both `ANY` and `REST` are Java classes predefined by PROSE. The matching-semantics of the method on line 3 depends on the crosscut class in which it is declared (`FunctionalCrosscut`) and on its signature. Because of the general signature, `doAct1` matches the method boundaries of all methods in all classes.

To restrict the number of join-points of the `doAct1` crosscut, its scope is narrowed (lines 5-7). The goal is to execute

¹The *crosscut object* programmatic construct in PROSE corresponds to a *pointcut* and an associated advice in AspectJ terminology.

```

1 aspect ExampleAspect {
2     pointcut doAct1(Printer p):
3         target(p) && call(* cancel(..));
4     before (Printer p): doAct1(p)
5         { // access control code }
6     pointcut doAccnt (byte[] b):
7         call(Printer.print(byte[] b));
8     after (byte b): doAccnt(b)
9         { // accounting code }
10 }

```

Figure 3: The AspectJ counterpart of the PROSE aspect in Figure 2.

the body of `ANYMETHOD` just upon entries of methods whose name matches "cancel". Specialization is achieved using the `FunctionalCrosscut.setSpecializer` method. A specializer object is a predicate that specifies a restriction of the default join-points defined by a crosscut. A further example of specializers shows how to restrict the `doAccnt` crosscut to classes inheriting `Printer` (lines 12-14). The specializer passed to the `doAct1` crosscut on line 5 is a logical-AND composition of two predefined PROSE specializers, namely `MethodS.BEFORE` and `MethodS.named()`. Specializers may be combined using OR operations as well. Both operations allow a flexible construction of specializer objects out of predefined building blocks.

Because aspects are defined in PROSE as classes, aspect instances are objects. An aspect instance contains the information about the matched join-points as well as the advice action and may be stateful like any other object.

2.2 AspectJ counterpart

The AspectJ counterpart of the PROSE aspect defined in Figure 2 is presented in Figure 3. Note that in AspectJ the advice declarations on lines 4 and 8 with keywords like `before`, `after` are language-specific and define *how* the crosscut is woven into the original code. In PROSE, the same effect of executing actions upon method entry can be achieved by specializing `doAct1` with the specializer `MethodS.BEFORE`. Because AspectJ is a distinct language, the aspect definition is more concise and uses operators for defining a pointcut (line 3) and wildcard constructs like "*" (line 3).

3. AOP IMPLEMENTATION

Current AOP implementations are based on transformations performed either on the source or directly on the object code. This section first explores AOP implementation alternatives for Java, then presents the current implementation of PROSE and evaluates the trade-offs between the two.

3.1 Compile-time approaches

To perform the weaving at compile-time, the code of the original application together with the aspect code can be merged into a new code version that includes the additional functionality. This approach either uses pre-processing or integrates the aspect weaver and the source compiler. The modified code participates in the build process and the new application is then deployed. Aspect instances and objects come to life simultaneously at application start-up. Compile-time AOP produces well-formed programs, since the resulting code must pass a compiler. Therefore, few additional

checks must be performed at run-time². The compiler may even optimize the resulting code, leading to improved performance.

An example of an AOP tool that works at compile-time is *AspectJ*. Its implementation is based on a language-sensitive pre-processor called *ajc*. It allows the definition of programming constructs that can be used to weave additional definitions or debugging functionality through a component’s code.

With AspectJ in particular, and compile-time techniques in general, the pointcuts are fine-grained. They require to have access to the source code and are closely tied to the programming language (in the case of AspectJ, Java).

3.2 Load-time approaches

Adding functionality to existing components available only in binary format can be achieved using *code instrumentation*. Approaches like *Binary Component Adaptation* (BCA) [7] and *Java Object Instrumentation Environment* (JOIE) [6] use this technique to transform an application at load-time. Both replace the class loader of a Java Virtual Machine (JVM) and change the classes at load-time.

BCA has been designed to solve problems related to component evolution and integration. BCA uses a *delta file* to specify load-time transformations of Java byte-code. A delta file is similar to Java source code except for a few additional keywords. BCA understands transformations like method or field renaming, additions, deletions, etc. The virtual machine executes the transformed code as if it would have contained the new or renamed methods in the first place.

Both BCA and JOIE do not require the source code. On the other hand, the transformations are necessarily class-related, the unit of extension being the class. Therefore, crosscutting concerns are difficult to express using BCA or JOIE.

3.3 Run-time approaches

In general, run-time application changes can be achieved using meta-object protocols (MOPs) [9, 5, 11, 15]. MOPs have been used to express cross-cutting concerns before the notion of AOP was proposed. In a reflective run-time environment, one can weave aspects through an application by locating the support for weaving and executing aspects directly in the executing environment. To match the AOP goals, run-time weaving should hide the complexity of the meta-level protocol. The AOP/ST [4] weaver follows this idea and provides run-time weaving for Smalltalk. In the context of adaptive programming, the DJ Framework [16] uses reflection to provide dynamic traversal strategies for Java.

Java, like Smalltalk, defines its semantics in terms of byte-codes that can be *interpreted* by an appropriate byte-code interpreter – the JVM. A straightforward run-time approach to AOP for Java is to locate the support for weaving and unweaving aspects directly in the JVM. An aspect-enabled JVM changes the execution model of an application. For each executed instruction, if the program counter points to a join-point, the interpreter executes additional actions related to the woven code. To do this, the JVM must also provide an interface for weaving aspects at run-time, which we call the Java Virtual Machine Aspect Interface (JVMAI). Unlike in the compile-time approach, aspects and applica-

²For certain crosscuts, e.g., those that refer to the control flow, run-time checks are needed.

tion may come to life independently. Aspect instances can be created externally to the targeted application and then be woven at an arbitrary point in time. In the JVMAI scenario, the main operation is to insert an aspect instance into a JVM. In the JVMAI context, the terms aspect insertion and weaving are used interchangeably.

Several design choices are feasible for the JVMAI. E.g., AOP/ST provides an interface that deals directly with aspect instances. A lower level interface can be located at the join-point level. This interface could define the way to create, register and unregister join-points and corresponding actions.

3.4 The PROSE JVMAI

A first JVMAI-based implementation of PROSE is designed as a JVM plug-in. Its prototype implementation is based on the debugger interface of a JVM, the Java Virtual Machine Debugger Interface (JVMDI) in Java 1.2 SDK (Java 2). The JVMDI is a low-level, native interface that allows a user to register requests for execution events inside a JVM and control execution for each event notification. It provides the means for inspecting the state of the JVM. The current implementation of PROSE is not tuned for performance but rather intended to provide a first implementation of run-time AOP and to create a platform for research on system issues of the JVMAI.

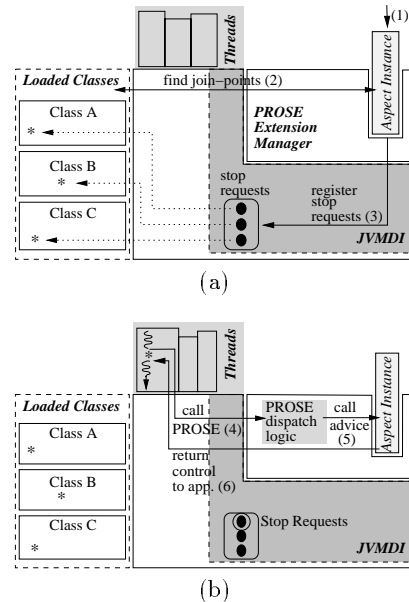


Figure 4: (a) PROSE actions at insertion-time. (b) PROSE actions at run-time.

Aspect insertion in PROSE is depicted in Figure 4.a. An aspect instance can be inserted into PROSE (1) using an interface called **ExtensionManager**. The extension manager is PROSE’s realization of the JVMAI. Immediately after insertion, the PROSE core inspects the JVM and then inspects the newly inserted aspect instance to find the set of join-points (2). For every join-point it requests a specific notification from the debugger layer in the JVM (3). The callback functionality for the join-point is also registered in the PROSE core at insertion-time.

Once a thread t reaches one of the registered join-points

(Figure 4.b) the execution of t is temporarily suspended, and the debugger passes the control of execution to PROSE (4). At this point, the PROSE dispatching logic calls the functionality in the crosscuts of the aspect instances that registered the join-point (5). During this step, PROSE inspects the current thread stack and passes the gathered information to the advice methods of the crosscuts. After the execution of advices is completed, PROSE returns the control to the application (6). Note that PROSE handles class load/unload events from the JVMDI to add and remove stop requests.

Aspect instance insertion

The aspect instance can be inserted either locally, from within an application running on the same JVM, or remotely sent to the extension manager. The following code fragment shows how local insertion takes place.

```
1 Aspect asp = new ExampleAspect();
2 Prose.extensionManager().insert(asp);
```

A new instance of the class `ExampleAspect` is created on line 1. Line 2 inserts the aspect instance `asp` in the extension manager. Immediately after line 2, PROSE starts trapping execution events and dispatching them to the crosscuts of the aspect-instance `asp`.

Alternatively, aspect instances may be sent to PROSE using a remote interface of the extension manager. In this case, the aspect instances are truly generated and initialized outside the JVM in which they will be woven. They are encoded in a serial format (e.g., marshaled) and sent over the network to the extension manager of the targeted JVM.

Implementation aspects related to JVMDI

A rich set of execution events can be expressed and intercepted using JVMDI, including field access and modification, catch and throw of exceptions, class loads and unloads, and breakpoints. Considering the number and type of events that can be requested and reported, the debugger interface is almost as powerful as the behavioral reflection mechanisms used in MOPs for Java [11]. Thus, one could use this interface to implement more AOP features, for instance an equivalent of the `cflow` pointcut designator in AspectJ. Currently, PROSE supports a restricted form of `cflow` specializers.

3.5 Design and implementation

Consequences of the source language approach

PROSE provides a set of pre-defined libraries for basic aspect-orientation: specializers of crosscuts depending on method names, class names, class inheritance relationships, member modifiers, or method boundaries. It also contains crosscut types for trapping field access and modification as well as for handling exceptions.

An aspect language should give programmers the means to extend the set of existing constructs and reuse existing programs. Current languages comply with this requirement and are fairly easy to extend. AspectJ, e.g., is an extension of Java and uses inheritance to facilitate reuse of aspects.

Choosing Java as aspect language eliminates the hard-coding of constructs in a separate language definition. The reuse or extension of aspect constructs is achievable by standard mechanisms of the source language. As an example, `ANY` or `REST` (Figure 2) are simple Java classes that extend a

```
public interface class Wildcard {
    // true if this wildcard matches the class cls
    public boolean isAssignableFrom(Class cls);
    // true if this wildcard matches the parameter list clsList
    public boolean isAssignableFrom(Class[] clsList);
}
```

Figure 5: Definition of the class `Wildcard`.

```
public abstract class CrosscutSpecializer {
    // true if the join point eR should be registered
    boolean isSpecialRequest(JoinPointRequest eR);
    // true if the join point event xEvent should be dispatched
    boolean isSpecialEvent(JoinPointEvent xEvent);
    // default implementations for AND, OR
    ...}
```

Figure 6: Definition of the class `CrosscutSpecializer`.

`Wildcard` superclass (Figure 5). By implementing its methods, one can define new wildcard types.

The idea of extending the aspect capabilities using standard Java mechanisms holds also for specializer classes. The method `named(String regexp)` in the class `MethodS` (Figure 2, line 6) returns specializer instances that restrict the scope of the crosscut to methods matching the given regular expression. The predefined specializer classes extend the `CrosscutSpecializer` abstract class described in Figure 6. The first method, `isSpecialRequest`, is called by PROSE at insertion time. Its parameter is a description of a join-point generated by the crosscut the specializer instance belongs to. The method checks whether the specified join-point should be registered or not. The second method (`isSpecialEvent`) is called by PROSE when the control flow reaches a join-point. If the method returns `false`, the advice of the crosscut the specializer belongs to will not be called.

An example of a customized specializer is `TIME_OF_DAY` that restricts the execution of advice methods corresponding to join-points reached within the specified time interval (Figure 7). The method `isSpecialEvent` returns `false` if the current time of the system is outside the desired time span (line 5). The `TIME_OF_DAY` specializer may be used, e.g., to allow aspect functionality to be executed just during a system's backup between 1 a.m. and 2 a.m. In a similar way, problem-specific specializers can be added.

Migration of code from the component to the aspect code is more restrictive on PROSE than on a compile-time AOP platform. The reason is that aspects, which are pure Java classes, can access only the public interfaces of the base component. The compile-time approach allows the advice code to reference the whole local environment of the join-point.

```
1 class TIME_OF_DAY extends CrosscutSpecializer {
2     public TimeInterval timeSpan;
3     // true only if the current time is in the specified interval
4     public boolean isSpecialEvent(JoinPointEvent xEv) {
5         if (currentTime().in(timeSpan))
6             return true;}
7 }
```

Figure 7: A `TIME_OF_DAY` specializer.

To overcome this shortcoming, one must use reflection. Note that this restriction does not apply in the opposite direction. Advices can be called from join-points corresponding to non-visible static points of a base class.

The obvious consequence of the source language approach is the trade-off between learning a library and learning the aspect language and the tools.

Consequences of the JVMAI model in PROSE

Aspects do not have to be created before the application starts. One can start writing an aspect class from scratch, instantiate it and insert it into a virtual machine already executing a given application. The aspect immediately starts changing the application. This scenario works without previous knowledge of the classes loaded by the JVM. At application build-time it is difficult to know all the classes that will be dynamically loaded by an application. Using AOP through the JVM allows the specification of aspects on all potential classes loaded by the JVM not just on sources known before application start. Later on, aspects can be withdrawn leaving the system in its original state.

Aspects can be efficiently tested by repeatedly inserting the aspect instance, checking the behavior of the application and eventually withdrawing the aspect instance to perform corrections. This method is helpful if the aspect is related to run-time parameters (e.g., system load). With a compile-time tool, the weaving of a new aspect version implies shutting down the system, thereby potentially loosing run-time data. Eventually, once the aspect code is stable, the aspect can be translated into a specialized aspect language and then be woven through the application code using a compile-time weaver.

The JVMAI approach is useful if behavior must be enhanced at run-time. If an exceptional condition occurs in a system, one could create and insert on-the-fly an aspect instance that logs relevant actions. Parts of a system's functionality may be subject to change over time (access control rights in a distributed system). A system administrator could remotely distribute aspects that control the access control policy into several nodes of an application without having to stop and restart the system.

From the performance point of view, aspect compilation and object-instrumentation have an *inlining* flavor. In contrast, deciding what advice functionality should be called at a certain join-point, checking type correctness, and performing variable conversions is done in the JVMAI model before actually executing the code at the join-point. Thus, JVMAI prevents code expansion but *interprets* join-point dispatching, thereby incurring a performance loss.

Finally, the JVMAI approach does not support crosscuts that add new members to a given class in the original code, because its implementation cannot change the source-code or byte-code of the original application.

3.6 Tool support

AOP platforms like AspectJ or Hyper/J provide powerful tools that let programmers see how a given aspect will affect their program at development time. Given dynamic weaving and dynamic class loading, it is important to know what aspects are currently inserted into an application and what join-points they actually denote. For this purpose, PROSE contains an Aspect Monitor tool. A screen-shot of the monitor tool is shown in Figure 8. The monitor displays a tree-like structure of aspects currently in-

Mode	Variables	Methods	Exceptions
debug mode	7.486 (s)	34.172 (s)	21.753 (s)
normal mode	1.484 (s)	14.324 (s)	11.072 (s)

Table 1: Debug-mode comparative benchmarks.

Measurement	PROSE	Time (ms)	StdDev (%)
hard-coded advice	no	0.001092	0.57%
advice (no args)	yes	0.076192	0.99%
advice (args)	yes	0.150479	0.43%
Breakpoint stop	no	0.026091	1.00%

Table 2: JVM performance with PROSE.

serted in PROSE, what crosscut objects they contain, and what join-points in the running application are matched by each crosscut. In this case, an instance of the aspect described in Figure 2 with its crosscuts is displayed. The aspect definition in Figure 2 is too general and matches the method `cancelLatestCommittedText` declared in a non-printer class. This aspect can then be withdrawn, be specialized to match just `Printer` methods, and be re-inserted.

4. PERFORMANCE OF RUN-TIME AOP

The JVMDII approach induces a certain overhead in the execution of an application. This section first analyzes the overhead incurred by the debug-mode of the JVM. It then concentrates on measurements specific to PROSE and finally points out possible improvements. All measurements are performed using Sun Microsystems JDK 1.2.2 Virtual Machine for Linux, using a 600MHz Pentium II system.

4.1 Debug mode micro-measurements

For obvious reasons, a JVM runs slower than normal in debug mode. For AOP purposes, the slowdown factor of join-point relevant operations is interesting: variable access, method calls and exception handling. Table 1 contains the results of running a benchmark [3] on Sun's JVM in debug mode (first line) and without debug mode (second line). The figures denote the number of seconds for every test. The observed performance loss is around a factor of two for exception throwing, while variable access is five times slower in debug mode.

4.2 Jspool application measurements

In practice, not all parts of the execution of an application are subject to the overhead imposed by the debugger. A further suite of measurements uses Jspool, a third-party component, as an example. Jspool is a distributed printing system for intranet use that consists of several services implemented as RMI objects. Jspool is entirely written in Java, is a non-trivial application (over 25000 lines of code, 180 classes), and is in operation since 1998. The experience gathered in connection with Jspool shows that the access control issue cuts across the whole system functionality. We implemented the access control policy of Jspool using dynamical PROSE aspects.

Since PROSE requires the JVM to run in debug mode, we compare the execution time of Jspool on a JVM running in debug mode against the time needed on a normal JVM. The debug mode increases the execution time of Jspool by 26%.

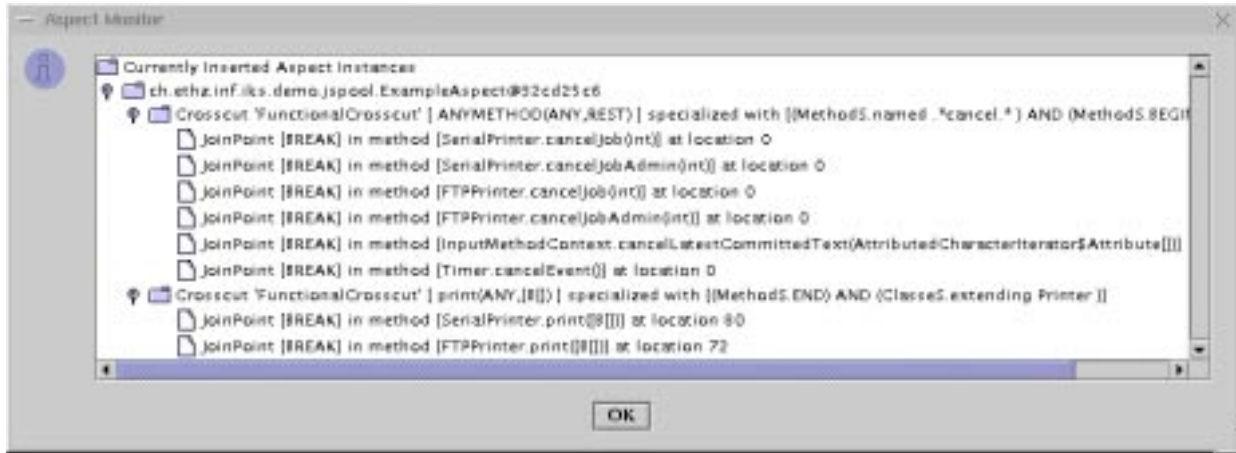


Figure 8: The JVMAI Monitor.

The performance of Jspool was measured once again with PROSE enabled and an aspect inserted into the extension manager. The aspect executes a void function for every incoming remote method invocation (RMI) of Jspool’s services. The aspect-enhanced version needed 14.845 seconds for 100 jobs against 14.746 seconds needed by the debug mode version of Jspool. The slowdown caused by PROSE with the inserted aspect is – compared to Jspool running on a JVM in debug-mode – 0.6%. Note however that other implementations of the JVMAI, not based on the debugger interface, are possible.

4.3 PROSE micro-measurements

To estimate the overhead of PROSE, we compare the performance of a JVM running in debug mode with and without PROSE activated and using aspects. These tests involve the execution of local calls.

The experiments measure the time needed to call a local method together with an advice action at one of the methods boundaries. Both the local method and the advice actions are empty. In all measurements, a join-point was hit 16000 times. To simulate a “typical call”, the invocation of the local method has two arguments: a (non-null) object and an integer value. A measurement round contains 16000 iterations. The measurements encompassed 160 rounds. The amount of memory available to the JVM is 32 Megabytes. The results are summarized in Table 2.

The first measurement (line 1) hard-codes the advice functionality at the beginning of the local method. This measurement approximates the cost of compiled advice on its own. The second measurement (line 2) shows the time needed by PROSE to call the advice functionality when the method boundary is reached. The third measurement (line 3) denotes the time needed by PROSE to call the advice functionality *and* to retrieve local variables from the stack. The fourth experiment measures how much time is spent in PROSE and how much time in the JVM. Line 4 shows the time needed by the JVM if it stops at the code location where the advice method should be called but then immediately continues execution.

Note that just the stop at a breakpoint location takes approximately 24 times more time than the execution of a hard-coded advice method. The advice functionality may

not always need to access local variables at the join-point, but if it does, the stack access performed by JVMDI to retrieve variables local to the join-point is slow (0.0371 ms). This cost can be computed by subtracting the time needed by PROSE to call an advice functionality that does not access the stack (line 2) and the time needed to execute an advice that performs two stack accesses (line 3) and then dividing it by two.

Once the join-point is reached, PROSE spends some time establishing the correct advice to be called, calling the advice and executing the body of the local method. This time (0.05 ms) can be computed as the difference between the time spent at a breakpoint (line 4) and the time needed per local call in the second experiment (line 2).

4.4 Performance discussion

As expected, the implementation of PROSE using JVMDI leads to an increase of the time needed to call an advice compared to advices woven at compile time. In the current implementation, the PROSE system itself is written in Java. Therefore, the aspect dispatch logic is interpreted.

For further experimentation with dynamic weaving and providing a useful prototype, a speedup is needed. The solution is to locate the JVMAI directly in the (native) core of the JVM. Additional measurements were performed to get an estimate of the speedup gained by a faster (native) dispatching mechanism for PROSE. The dispatching functionality was simulated once in the presence of a Just-In-Time compiler (JIT) and once with the JIT disabled³. Unlike the measurements presented so far, the simulation of join-point dispatching uses the version 1.3 of Java 2 SDK (which has better JIT support for Linux). The platform used for the simulation measurement is different than the one used for the PROSE measurements, however the goal of this measurement is to find the relative speed gain induced by the JIT. In the simulation, the dispatching logic was 9.1 times faster when it is just-in-time compiled. The result gives us confidence that the current implementation can be significantly improved by using a mechanism similar to just-in-time compiling.

³For all other tests, the JIT compiler was de-activated.

5. CONCLUSIONS

The PROSE system provides a homogeneous platform for prototyping and testing AOP applications in Java. For creating aspects, one writes pure Java classes based on the PROSE library. The aspect code can be compiled using a standard Java compiler and woven through an application at run-time. Repeated weaving and unweaving of aspects speeds up aspect testing and validation. If special constructs (otherwise hard-coded in a language definition) are needed, they can be added by extending the PROSE framework.

The current performance limitations make PROSE less appropriate for computational-demanding applications. However, PROSE can shorten the development phase of such applications, and can help developing systems that require dynamic weaving because of environment changes. The measurements made give us confidence that the current performance can be improved by combining the run-time approach with other techniques like just-in-time compiling.

6. ACKNOWLEDGMENTS

We thank to the System Programming Group at the Technical University of Darmstadt, especially G. Herr and Prof. H. Walschmidt, for generously allowing us to use Jspool for the experiments with PROSE. We appreciate discussions with the participants of the research seminar on system evolution in the Spring 2000 at ETH.

7. REFERENCES

- [1] M. Aksit and M. Bergmans. Composing Synchronisation and Real-Time Constraints. *Journal of Parallel and Distributed Computing* 36, pp. 32-52, 1996.
- [2] M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting Object Interactions Using Composition Filters. *Lecture Notes in Computer Science*, 791:152, 1994.
- [3] D. Bell. Make Java fast: Optimize! *JavaWorld: IDG's magazine for the Java community*, 2(4), Apr. 1997.
- [4] K. Bollert. On Weaving Aspects. Position paper at the ECOOP'99 workshop on Aspect-Oriented Programming, June 1999.
- [5] S. Chiba. A Metaobject Protocol for C++. *ACM SIGPLAN Notices*, 30(10):285-299, Oct. 1995.
- [6] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic Program Transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167-178, Berkeley, USA, June 15-19 1998. USENIX Association.
- [7] R. Keller and U. Hölzle. Binary Component Adaptation. In *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307-329. Springer, 1998.
- [8] P. Kenens, S. Michiels, F. Matthijs, B. Robben, E. Truyen, B. Vanhaute, W. Joosen, and P. Verbaeten. An AOP Case with Static and Dynamic Aspects. In *ECOOP'98 Workshop on Aspect-Oriented Programming*, Brussel (Belgium), July 1998.
- [9] G. Kiczales and J. des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.
- [10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *1997 European Conf. on Object-Oriented Programming (ECOOP '97)*, pages 220-242. Springer Verlag, 1997.
- [11] J. Kleinoeder and M. Golm. MetaJava — A Platform for Adaptable Operating-System Mechanisms. In *Object-Oriented Technology: ECOOP'97 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science*, pages 507-514. Springer, 1997.
- [12] K. Lieberherr and L. Mezini. Aspect-Oriented Components. *Technical Report*, College of Computer Science, Northeastern University, Boston, MA, 1999.
- [13] C. Lopes. D: A Language Framework for Distributed Computing. Ph.D. Dissertation, College of Computer Science, Northeastern University, Boston, 1997.
- [14] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. *ACM SIGPLAN Notices*, 33(10):97-116, Oct. 1998.
- [15] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and F. Kimura. OpenJIT : An Open-Ended, Reflective JIT Compiler Framework for Java. In *Proceedings of ECOOP'2000*, Springer Verlag, 2000.
- [16] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag.
- [17] A. Popovici, T. Gross, and G. Alonso. Aop support for mobile systems. Paper at the OOPSLA'01 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Oct. 2001.
- [18] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *1999 International Conference on Software Engineering*, pages 107-119, Los Angeles, CA, USA, 1999.
- [19] E. Truyen, B. Joergensen, and W. Joosen. Customization of Component-Based Object Request Brokers through Dynamic Configuration. In *TOOLS Europe'2000*, IEEE Press, pages 181-194, June 2000.
- [20] B. D. Win, B. Vanhaute, and B. D. Decker. Towards an Open Weaving Process. Position paper at the OOPSLA'01 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Oct. 2001.
- [21] Xerox Corporation. The AspectJ Programming Guide. Online Documentation, 2001. <http://www.aspectj.org/>.