Apostle: A Simple Incremental Weaver for a Dynamic Aspect Language

TR-2003-16

Brian de Alwis
bsd@cs.ubc.ca

Gregor Kiczales
gregor@cs.ubc.ca

Dept. of Comp. Sci.
University of British Columbia
Vancouver, Canada

ABSTRACT

This paper describes the incremental weaving implementation of Apostle, an aspect-oriented language extension to Smalltalk modelled on AspectJ. Apostle implements incremental weaving in order to make aspect-oriented programming (AOP) a natural extension of the incremental edit-run-debug cycle of Smalltalk environments. The paper analyzes build dependencies for aspect declarations, and shows that two simple dependency table structures are sufficient to produce reasonable re-weaving efficiency. The resulting incremental weaver provides re-weaving performance proportional to the change in the program.

1. INTRODUCTION

This paper describes the incremental weaving implementation of Apostle [4], a general-purpose aspect-oriented extension to Smalltalk [6] modelled upon AspectJ [1, 10]. The Apostle project leverages the well-known AspectJ language semantics in order to uncover the requirements of adding AOP support to a Smalltalk environment, while preserving the character of the development environment so as to not change the expected programming work-flow.

Not all AspectJ constructs map perfectly to Smalltalk. Smalltalk is a dynamically-typed language. This means that the type of an object, such as an argument to a method, is determined exclusively at run-time. This complicates implementing AspectJ pointcuts such as *target()* and *call()*.

Smalltalk is also typically programmed in environments that promote incremental development, which are themselves written in Smalltalk [5]. In these environments a programmer edits at the level of individual elements, such as a method or class description, and changes are immediately installed and reflected in the running system. The system need not be type consistent to be able to run: run-time errors are manifested as *walkbacks* in the debugger, and are often fixed on the fly.

This *liveness*, or immediacy, is impossible to maintain with the *whole-program weaving* techniques used by most AOP language implementations, where even a single change would require re-weaving the entire program. For example, IBM VisualAge for Smalltalk 5.5 (VA/ST), the Smalltalk development environment targeted by Apostle, ships with over 3000 classes in its standard image: full recompilation would be time consuming. AOP language implementations for such environments must seek to incorporate incremental weaving techniques to minimize re-weaving times and maintain the expected liveness.

Apostle shows that the AspectJ model of AOP can be made a natural extension of Smalltalk, with a reasonably efficient incremental implementation that maintains the incremental edit-run-debug cycle of Smalltalk environments. As such, this paper makes two contributions. The first is that a usable AOP extension to Smalltalk can be obtained by eliminating the AspectJ join points that do not appear in Smalltalk, and simplifying the pointcut mechanisms to eliminate any static type testing. The other contribution shows that two simple dependency table structures are sufficient to produce reasonable re-weaving efficiency.

Incremental weaving is a critical issue for AOP — not just for exploratory or interactive programming environments, but for any environment where code may be added or modified over time, rather than presented once, such as dynamic code-loading.

This paper proceeds as follows: Section 2 describes other attempts to create AOP languages for Smalltalk and other incremental AOP solutions. Section 3 demonstrates the Apostle language through two examples. Section 4 describes the Apostle weaver implementation. Section 5 summarizes some of our experiences and interesting results from the implementation, as well as future work.

2. RELATED WORK

There have been several other attempts to create AOP language extensions for Smalltalk, though none support incremental weaving. The AspectJ Development Tools currently support a limited form of incremental weaving. These are described here.

AOP/ST [2, 3] is a straight-forward port to Smalltalk of AspectJ 0.1, which provided support for building domain-specific languages. AOP/ST presents a pure Smalltalk binding, meaning that it expresses its capabilities through Smalltalk messages and does not introduce any special language constructs. It also requires a separate compilation/weaving step; there is no support for incremental weaving.

Andrew [7] is another AspectJ-style AOP language for Smalltalk. Andrew replaces the AspectJ-style pointcut language with a logic meta programming language, extended with certain predicates so as to form a new pointcut language. By using a logic meta programming language, the user can extend the pointcut language with new pointcut types. Andrew's implementation follows a *static weaving* model, as does Apostle, where join points are mapped to feasible locations in the code and which are transformed to dispatch advice. Andrew requires an explicit weaving step, and as such does not support incremental weaving.

AspectS [8] has focused on the creation of a metaobject protocol (MOP) for investigating AOP issues in Squeak [14], another Smalltalk variant. Its join point model targets only method executions. AspectS has no distinct pointcut language: the programmer explicitly identifies methods, usually derived using Smalltalk's reflection. Advice uses *qualifiers* to discriminate on dynamic properties of the join point. Advice must be explicitly explicitly installed and removed, and as such, does not support incremental development. As there is no separation between the identification of join points and the advice, there is no manner for the system to incrementally add or remove from the identified shadows.

The AspectJ Development Tools (AJDT) [15] extends the Eclipse development environment [13] with AspectJ support. The AspectJ compiler supports incremental weaving, and as such is the closest comparator to Apostle. However it rebuilds the entire project should any aspect change.

3. THE APOSTLE LANGUAGE

The Apostle language is derived from AspectJ 1.0 [10, 11], featuring aspects, pointcuts, advice, and a dynamic join point model. Table 1 summarizes these features, and contrasts them with those of AspectJ. Apostle also provides a metaobject protocol (MOP). In the interests of space, we demonstrate Apostle through two examples in Section 3.1. Given the similarity to AspectJ, this should suffice to convey the general feel of the language. Differences from AspectJ are described to Section 3.2. A more thorough description of the language, including the MOP, is available elsewhere [4].

3.1 Examples of the Apostle Language

Consider the following example, where a programmer would like to be alerted when the coordinates of a *Point* change.[1] The coordinates can only change by using a *Point*'s setters, the methods *Point*≫*#x:* or *Point*≫*#y:*. We first create an aspect, as advice are housed on aspects. Aspects are created in a similar manner to Smalltalk classes, by sending a message to their super-aspect; all aspects are rooted from *ApAspect*:

```
ApAspect subaspect: #MoveNotification
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

We next define a pointcut *moves()* which identifies all join points where the receiver is a *kind of Point*, meaning an instance of the type *Point*, and is executing a method corresponding to *#x:* or *#y:*.

[1]This example is intentionally similar to the one used in [10], which provides a good introduction to AspectJ-style AOP as well as a detailed explanation of this example.

```
pointcut moves(): kindOf(Point) & (executions(#x:) | executions(#y:))
```

Finally, we define advice to notify the programmer after any such change:

```
after<moves()>

    Transcript cr; show: 'Point moved'
```

The next example uses *around* advice to wrap string queries to a registry (an associative array) to use a case-insensitive key. PROCEED is Apostle's analogue to AspectJ's proceed() with no arguments, and PROCEEDUSING to use of proceed(...) with argument replacement. PROCEEDUSING exploits Smalltalk's keyword message syntax to allow selective argument overriding:

```
pointcut fetches(reg, key): reception(#at: key) &
    kindOf(MtRegistry reg)

around<fetches(reg,key)>

    key isInteger ifTrue: [^PROCEED].
    ^PROCEEDUSING key: (MtCaseInsensitiveWrapper for: key)
```

Note that neither reg nor key are statically typed, as expected for a dynamically-typed language.

We call *moves()* a *named* pointcut: it exists independently of advice. The advice could have been defined specifying the pointcut definition directly, as in:

```
around<reception(#at: key) & kindOf(MtRegistry reg)>
```

We call such a pointcut an *anonymous* pointcut, as it cannot be referred to by name.

3.2 Language Differences from AspectJ

The differences between Apostle and AspectJ, contrasted in Table 1, can be categorized into one of four classes.

Non-Critical Elements

Apostle was intended to be a simple language with just enough features to enable our experiment. These differences are language features omitted as they were deemed non-critical. Examples include AspectJ's inter-type declarations and pointcuts such as *cflowbelow()* and *if()*.

Two other major omissions are left for future work. While aspects are inherited in Apostle, advice is not. Smalltalk does not have the concept of abstract classes or methods — they are identified only by convention. The desired behaviour is not certain. Nor does Apostle provide for support for targeting method call join points. While possible, Smalltalk's dynamic typing makes this extremely difficult. Method call join points can be somewhat approximated using method reception join points by advising the reception of the called message.

Inapplicable Elements

Apostle's join point model consists of only two types: the reception of a *message* by an object, and the execution of the methods corresponding to that message. AspectJ provides for identifying conditions that have been formalized as language elements in Java, such as constructors, exception catching, initialization and static initialization. Although these same conditions exist in Smalltalk, they are not formally defined and are indistinguishable from normal methods.

Language Elements	**Apostle**	**AspectJ**
Join points	method executions, message receptions	method executions, method calls, constructors, exception handling, initialization, static initialization
Pointcuts	*execution(), reception(), kindOf(), sender(), cflow(), type()*	*execution(), target(), call(), cflow(), cflowbelow(), this(), within(), withincode(), initialization(), staticinitialization(), get(), set(), handler(), if(), args()*
Advice	*before, after, around*	*before, after, after throwing, after returning, around, around throws*
Aspect types	singleton, per-cflow, per-object	singleton, per-cflow, per-cflowbelow, per-this, per-target
Join point context	thisJoinPoint	thisJoinPoint, thisStaticJoinPoint
Inter-type declarations	none	introduction, declare parents, declare warning, declare error, declare soft

Table 1: Comparison between language elements of Apostle and AspectJ.

Alternative Existing Implementations

Some constructs are omitted as they can be implemented using existing constructs. Apostle does not provide distinct get-instance-variable or set-instance-variable join points; since the instance variables of a Smalltalk object cannot be directly accessed outside of the object except through the use of accessor methods, which can be captured. Being just methods, however, these join points can be captured by treating them as normal methods. Other AspectJ pointcuts have been cast as different pointcuts in Apostle, or have an equivalent implementation.

Syntactic Changes

The final class consists of minor syntactic adjustments, made to more easily distinguish elements from normal Smalltalk elements like methods, or to make the language more Smalltalk-like. Thus the use of ‘<’, ‘>’ as advice pointcut-delimiters, or single ‘&’ and ‘|’ in pointcut composition.

4. IMPLEMENTATION

Apostle uses a source-to-source compilation strategy, weaving the advice and source into an equivalent pure-Smalltalk result called the *target model*. This is then compiled and installed using the Smalltalk compiler. This means that Apostle requires no modifications to the underlying virtual machine. For runtime efficiency, the implementation makes some assumptions which are embedded in its generated code.

Incremental weaving is to weaving as incremental compilation is to compilation: for any change to the program, only the necessary parts are re-weaved. The re-weaving is minimized by understanding and recording sufficient information on the assumptions embedded in the target model.

Section 4.1 describes the target model in brief. Section 4.2 then describes how the incremental weaver fixes the target model arising from some change. Some performance characteristics are summarized in Section 4.3.

4.1 Target Model

The semantics of the Apostle language requires executing advice whenever execution reaches a join point matched by its pointcut [12]. To statically weave the program, the Apostle weaver identifies all possible code locations corresponding to each join point (called the *join point shadows*), which are then statically modified to dispatch all advice matching the join point.

4.1.1 Join Points and Join Point Shadows

Apostle's join point model identifies points in the *run-time* object call graph, specifically method receptions and method executions. Even though join points are points in the execution, we can still identify places in the code that *might* correspond to some join point. These places are called *join point shadows*.

One consequence of this model is that some information associated with a join point shadow can be statically determined. For example, at the shadow of execution join points for *Point≫#x:* we know that the name of the method is *#x:*. Other properties are only known dynamically, such as the arguments to the method, or whether a reception shadow actually corresponds to a reception join point. There is a well-defined distinction between the properties able to be determined statically and dynamically at a join point shadow.

4.1.2 Determining Advice Applicability at Join Point Shadows

A join point shadow can be tested against a pointcut statically. In some cases, some further run-time checking is required to ensure a match/fail. We call the testing that can be done statically the *static testing*, and the run-time testing the *residual testing*. Table 2 summarizes the information tested by each of Apostle's primitive pointcuts, including which information can be statically determined. Shadows can be eliminated entirely from consideration if they cannot be statically matched by a pointcut. We define a shadow's *applicable advice* as the advice whose pointcut statically matches the shadow.

A residual test is simply code that tests the execution context in some manner. For example, consider the following pointcut:

```
execution(#at:) & cflow(execution(#lookup:))
```

As we can see from Table 2, *execution()* only performs static testing, and *cflow()* residual testing. Thus the *execution()* pointcut will select only those shadows for methods named *#at:*. The *cflow()* requires a residual test at those shadows to ensure the execution is resulting from executing a method *#lookup:*.

4.1.3 Transformation of Join Point Shadows: Dispatching Advice

Transformation of a shadow then entails in-lining the advice dispatch code for the applicable advice. This is conceptually expressed as:

Kind	*reception()*	*execution()*	*kindOf()*	*type()*	*sender()*	*cflow()*
sender					E	
sender class	T				T	
sender mname	T				T	
target			E	E		
target class			T (s)	T (s)		
method	T (s)	T (s)				
args	E	E			E	
process						X
is super send	T					

Table 2: The join point information tested (T), and possibly statically known (s), by Apostle's primitive pointcuts. Some parts of the context at the join point is able to be exposed (E) to the advice. *cflow()* uses process information to test if its condition has occurred, marked (X).

```
Point≫x: newX

    "Dispatch applicable before advice"
    (residual pointcut tests) ifTrue: [call before advice].
    (residual pointcut tests) ifTrue: [call before advice].

    x := newX   "the original method body"

    "Dispatch applicable before advice"
    (residual pointcut tests) ifTrue: [call after advice].
    (residual pointcut tests) ifTrue: [call after advice].
```

Since the advice at any shadow is known, the advice is simply listed one at a time; *before* and *after* advice are not required to be paired. The residual tests, themselves Smalltalk expressions, serve to confirm that this join point is matched by a pointcut. Apostle does not impose or provide any manner to alter the ordering of advice.

Dispatching *around* advice is slightly different from that of *before* and *after* advice. Should the advice apply, then *around* advice can affect — and even prevent — the remainder of the computation. But if the join point is not matched (i.e. the residual test fails) then the remainder of the computation must be executed. In Apostle, this computation-remainder is encapsulated as a Smalltalk block, a closure similar to a LISP lambda expression. Then *around* dispatch would conceptually be expressed as:

```
Point≫x: newX
    | proceedBlock |

    proceedBlock := [
        x := newX   "the original method body"
    ].

    "Dispatch applicable around advice"
    (residual pointcut tests)
        ifTrue: [
            call around advice providing proceedBlock
            "dispatched advice responsible for dispatching proceedBlock"
        ifFalse: [
            "join point not matched; continue with computation-remainder"
            proceedBlock value].
```

For a shadow identified by multiple *around* advice, the contents of the proceedBlock would have the next *around* advice dispatch. As a result, the advice traversal is fully encoded in the shadow. This allows the advice implementations (discussed next in Section 4.1.4) to be re-used and called from a number of join point shadows, reducing the compile-time cost when weaving advice.

4.1.4 Transformation of Advice

The advice bodies are transformed into normal Smalltalk methods.

```
unique_selector: thisJoinPoint object: _AP_receiver
    exposedContext1: exposedContext1 ...

    "advice body"
```

The selector name is chosen to be unique per piece of advice so as to be able to distinguish one piece of advice from another. This is necessary as the environment and language must have some unique way to distinguish advice since advice is not otherwise named. The issue is discussed further in Section 5.2.3.

The exposed execution context of the join point, including thisJoinPoint, is pre-computed at the shadow and provided to the advice as arguments, as in AspectJ.

4.1.5 Efficiencies in Residual Testing

Apostle's implementation of *cflow()* uses advice to cause a marker to be set at any join points identified by its sub-pointcut. This marker is later queried by the residual tests for the advice that actually specified the *cflow()*.

This process has been generalized such that pointcut implementations can provide *customizations* to the system, in the form of advice. *Cflow()* simply defines *around* advice targeting its sub-pointcut, whose advice-body causes the marker to be set for the duration of the join point.

4.2 Incremental Weaving

This section describes the state maintained by Apostle to implement efficient re-weaving of the target model after some change. We found that, having identified the dependencies in the Apostle target model, we can implement an incremental weaver which operates in time proportional to the extent of the change by maintaining just two dependency lists.

4.2.1 Dependencies Amongst Language Elements

The efficient processing of incremental weaving requires identifying the impact of any program change. Whether adding a method, changing a pointcut, or removing some advice, each change may introduce inconsistencies into the target model which must be subsequently repaired and reconciled. Minimizing the re-weaving requires identifying the assumptions and dependencies compiled into the target model and regenerating the minimal elements to fix any resulting breakage.

Figure 1 shows the dependencies and assumptions compiled into the target model. As can be seen from this model, all direct depen-

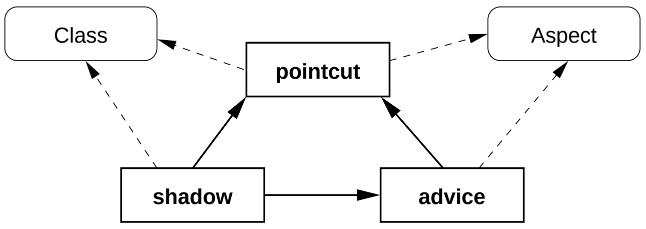

Figure 1: Dependencies existing between elements in the target model. Solid lines denote links implicitly embedded in the source by the weaver; dashed lines indicate links explicit in the code structure.

dencies can be traced eventually to the pointcuts. These dependencies are summarized as follows:

- *Advice are dependent on their pointcuts for identifying applicable join point shadows.* A change to an advice's pointcut may change the join point shadows matched, and thus require any previously- or newly-identified shadows to be regenerated.
- *Shadows have embedded calls to applicable advice and pointcut residual tests.*
- *Advice and pointcuts depend on the scope of their enclosing class or aspect.* A change in where a pointcut or advice is defined, whether to a different class or aspect, necessarily changes the scoping of any references from within the pointcut. Consider advice which used a pointcut inherited from a super-aspect, a pointcut which is subsequently overridden. This requires re-weaving the advice for any previously- or newly-identified shadows.

4.2.2 Recording Dependencies

As noted in Figure 1, some of the dependencies are recorded explicitly in the code structure. For example, advice and pointcuts are defined on aspects, and pointcuts and methods are defined on aspects.

The Apostle weaver records the implicit, embedded dependencies in two lists, one which records the advice-pointcut dependencies and the other the shadow dependencies. These two lists are sufficient to ensure minimal re-weaving. These tables are updated at each re-weaving of the image.

Pointcut Dependency List: This list records the advice dependent on each pointcut. A change to the definition of any of these pointcuts requires re-weaving all dependent advice. Advice is dependent on any pointcut reachable from its pointcut: the *transitive closure*. Consider the following example:

```
pointcut p1(): ...
pointcut p2(): p1() & ...

before<p2()>
    ...
```

A change to *p1()* may change the join points matched by *p2()*, and hence requires the advice be re-woven for any previously- or newly-identified shadows. Thus the advice is dependent on *both p1()* and *p2()* — and any their subpointcuts. A change to *any* of these pointcuts requires reweaving this advice.

This list grows in size with the number of *named* pointcuts and the number of advice. This is likely to be a sparse list, as not all advice will be dependent on every named pointcut.

Shadow Dependency List: This list records all advice applicable for each shadow, meaning the advice whose pointcut statically matches the shadow. This is used when re-generating the shadow, as occurs when advice is added or removed, or the corresponding method is redefined.

This grows in size with the number of shadows and the number of advice applicable at each shadow.

We also maintain a third list of all the currently defined advice. This speeds determining which pieces of advice apply to newly-defined methods, as advice have a reference to their pointcut. This list is linear in size with the number of advice.

4.2.3 Types of Incremental Changes

There are several possible types of program change in the Smalltalk environment. Each has some impact on the target model, potentially invalidating some of the weaving. These are *individual* changes: many common changes, such as loading a class with already defined methods, are *compound* changes.

Adding a method: The method may be subject to advice.

Adding advice: All shadows at which advice applies require reweaving.

Adding a new pointcut: This may override a previously inherited definition.

Adding a class/aspect: In the absence of advice inheritance, this should have no impact on the target model.

Removing a method: The method may have been subject to advice.

Removing advice: All shadows dispatching the advice require reweaving.

Removing a pointcut: Any advice using the pointcut must be removed from the previously-identified shadows, which must be re-woven. Any pointcut customizations must be removed.

Removing a class/aspect: This is effectively the same as adding/ removing all methods, advice, and pointcuts defined by the class/aspects and its subclasses/subaspects

The number of changes is reduced with the observation that the modification of a program definition can be equivalently implemented as removing the old definition and adding the new definition.

To fix the target model, the weaver must obtain notification of every such change affecting the model. Apostle's implementation required using advice to achieve some of this notification. As described above, class redefinition is implemented as a remove-class, add-close combination. To effect this, Apostle requires notification that class-redefinition is about to be effected; VA/ST however, offers notification only once a class redefinition is complete. Apostle obtains this notification by targeting advice on methods related to class-redefinition.

4.2.4 The Incremental Weaving Algorithm

Apostle's re-weaving algorithm is structured and implemented for safety, meaning that the dependencies identified above are used to ensure pieces are not installed before their required dependencies are available. For example, advice implementations are installed before any shadows can possibly call it, and all modified shadows compiled before being collectively installed. This is important as there may be other processes executing in the background, whose execution may reach certain targeted join points.

The algorithm takes as input lists of the removed and added program elements. This process can be broken into three major steps, as follows:

Step 1: Process all removals: removed methods, advice, pointcuts, classes and aspects. At the end of this step, the target model will have been brought to temporary consistency in that all shadows are identified by current pointcuts, and only dispatch to existing advice.

Step 2: Perform the pointcut redefinitions.

Step 3: Process all additions: added methods, advice, pointcuts, classes and aspects. At the end of this step, the system will have been brought into consistency with all changes.

These steps are expanded below. Over the course of the algorithm, various shadows are scheduled to be regenerated because of added or removed advice. These modifications are accumulated and finally committed in one step rather than as individual changes, as is seen below.

Pointcuts are distinguished between *named* and anonymous pointcuts. Anonymous pointcut, those defined explicitly by advice rather than as a separate named pointcut, are treated as either added-pointcuts or removed-pointcuts should their defining advice be added or removed. Named pointcuts and must be explicitly removed.

The algorithm is then as follows:

Step 1: Process All Removals

All removals must be processed using the old pointcut definitions so as to find and fix-up the shadows previously-identified by either removed-advice or removed-pointcuts.

1. *Delete all records for shadows corresponding to removed methods from the* Shadow Dependency List.
2. *Remove any customizations used by removed pointcuts.* Some pointcuts are themselves implemented using advice for efficiency. *cflow()*, for example, uses advice to set a market at its sub-pointcut.
3. *Undo the effects of the removed pointcuts.* Use the old pointcut definitions to identify old shadows. Schedule these shadows for regeneration because of the removal of any advice using this pointcut.
4. *Commit the shadow modifications.* These shadows must be regenerated before any advice to be removed is uninstalled in the next step, so as to protect against any inadvertent dispatches to now non-extant advice.[2]
5. *Remove the removed-advice.* Remove the actual advice implementation methods.
6. *Delete the pointcut dependencies for any advice using a removed-pointcut from the* Pointcut Dependency List. The actual advice affected should be remembered, if they are not being removed; they may still be applicable due to added-pointcuts. These dependencies are rebuilt in Step 3-1.

At the end of Step 1, all dispatches to removed-advice and advice identified by removed-pointcuts have been by the regeneration of the relevant shadows in Step 1-4. The target model is now internally consistent.

Step 2: Perform Pointcut Redefinitions

The new pointcut definitions are made current. Pointcut deletions are treated as equivalent to redefining the pointcut to nothing.

Step 3: Process All Additions

This essentially mirrors Step 1.

1. *Update the* Pointcut Dependency List *for new and re-woven advice*. This includes the advice whose pointcuts were removed/redefined in the previous phase.
2. *Add required customizations for new pointcuts.* These customizations may require installing additional advice.
3. *Install the new advice.* Install the advice implementation methods. These must be in-place before re-weaving the modified shadows in Step 3-6.
4. *Use the new pointcut definitions to identify possible join point shadows.* Schedule these shadows for regeneration to incorporate dispatches to possibly applicable advice.
5. *Process the new and updated methods.* Test all new or updated methods to see if they are identified as join point shadows by any previously-defined pointcuts. Scheduling these shadows for regenerate to dispatch to any existing advice.
6. *Commit the shadow modifications.* These shadows are regenerated after the advice has been installed, so as to protect against any inadvertent dispatches to non-extant advice.

4.3 Evaluation

Apostle was implemented on VA/ST 5.5.2 for Linux. VA/ST is a representative Smalltalk system, featuring approximately 3000 classes in its standard image. This section provides some observations on the performance of the system.

[2]As might happen if another process' execution hits an identified join point. This requirement was identified during the development of this algorithm, where shadows were regenerated *after* the advice with unfortunate consequences — as the compiler had been targeted with advice. The next step (then) in the algorithm, regenerating the shadows, required the compiler and thus caused an old shadow to dispatch to now non-existent advice, subsequently causing a run-time error. But this error could not be backed-out, as trying to run any code-snippets required a working compiler.

Operation	Average Time
Identifying applicable shadows	
reception(#at:) (2449 shadows)	1.84s
execution(#at:) (50 shadows)	1.47s
reception(#at:) & kindOf(Array) (1 shadow)	1.46ms
Compilation times	
Smalltalk method, without Apostle	408μs
Smalltalk method, using Apostle	617μs
simple pointcut, using Apostle	272μs
before advice, using Apostle	4.83ms
around advice, using Apostle	5.41ms
Save and installation times	
single method, without Apostle	37.8ms
single method, targeted by no advice	37.8ms
single method with 20 advice	42.3ms
saving *and* deleting advice, advising 17 shadows	424ms
Changing class definitions	
leaf class, with Apostle	7.9s
leaf class, without Apostle	85.5ms
class with 2 subclasses, without Apostle	227.5ms
class with 2 subclasses, with Apostle	30.6s
Method and advice execution times	
minimum method execution time	0.08μs
minimal method execution time	0.45μs
minimum advised method execution time	2.08μs

Table 3: Usability Timings; these are described in Section 4.3.1

4.3.1 Performance Characteristics

Apostle's performance characteristics can be categorized in two parts: the times for performing weaving, and the resulting execution times. Table 3 shows some very preliminary timings for several operations: these were obtained using Apostle on a 400MHz Celeron running VA/ST 5.5.2 for Linux. Some explanatory notes:

- Two timings are quoted for method execution time. The Smalltalk compiler recognizes and optimizes certain method footprints, such as returning an instance, as specially executed *primitive* methods. Thus the minimum method execution time is such a primitive method, while the minimal method makes some non-garbage-creating calls.
- Compilation of an element is not the same as saving: saving compiles and installs the element.
- Class definition changes under Smalltalk requires recompiling all methods of a class and its subclasses, and are expected to take longer.

Most operations using Apostle complete in under 0.5 seconds, with two class of exceptions. The first class were those operations which could be expected to take longer, such as the single use of *reception()* or *execution()*, which require searching all classes in the image.

The other, and more notable, class involved class definition changes. Modifying the definition requires removing all advice from throughout the hierarchy, and then re-installing after the completion. In fact, Apostle's performance differs significantly from normal Smalltalk development when dealing with changes to any large class hierarchy, For example, adding a method to *Behavior* causes a noticeable pause, since these require processing *all* classes in the image — as all classes inherit from *Behavior*, upwards of 3000 — to see if advice targets this new method. This is an area requiring further work and optimization.

4.3.2 Qualitative Performance

Initial indications are that the Apostle implementation preserves the liveness of development in the system. Saving any definition causes the change to be immediately effected. The ordering in defining methods and pointcuts identifying those methods has the same net effect: the appropriate methods are properly targeted with the applicable advice. Aspects can be loaded and removed from the image, and are immediately woven or unwoven respectively.

Qualitatively, re-weaving response generally seems adequate. Saving methods is generally quick, as is saving advice with limited impact, as identified in Table 3, where such operations generally complete in under half a second. The exception in the noted case with large class hierarchies.

Apostle was used to restructure three small Smalltalk application, implementing aspects to handle diverse needs such as separating the subject/observer [9], optimizing graphics redrawing, serializing access to a shared object, adding progress monitoring to a long-running algorithm, and hooking into a closed framework in unanticipated ways; these are described in more detail in our dissertation [4]. All were implemented using small, localized aspects, and evolved over a series of trial sessions, with many saves and re-saves of methods, advice, and pointcuts. One benefit of the integrated incremental weaving was being denied any opportunity to forget an explicit weaving cycle before running the applications!

5. DISCUSSION

This section discusses some of the interesting results from the implementation, and possible future work.

5.1 Implementation Design Decisions

5.1.1 Trade-off Between Speed and Optimization: Encoding Advice Traversals

There is a trade-off between the re-weaving speed and faster execution. Faster execution can be obtained by making and embedding further assumptions throughout the target model. But this increases the complexity of the target model, as well as increasing the amount of work necessary to fix and re-weave upon any change.

Given the liveness requirements, we chose to concentrate on re-weaving speed, which also results in a simpler implementation. The method used for encoding advice traversals is one example of this.

There are two approaches to implementing advice, differing by where calls to advice are made. A piece of advice generally applies to a set of join points. A subset of those same join points may also be identified by some other pieces of advice. Thus different join points shadows may have different selections of advice, meaning that there are different traversal ordering of the advice on a per-join point basis. One can choose to embed the advice traversal into the advice or into the join point shadows.

Embedding the traversal into advice requires having customized copies of the advice for each possible traversal, where each copy is customized to embed the call to the next advice for that particular traversal. This results in a large number of copies of advice, but creates more opportunities for optimization. Apostle uses the other possibility, in which the advice traversal is embedded in the join point shadow. This reduces the amount of work necessary during a save.

Considering that incremental development often leads to a method to be changed many times before ever executed, one could see a two-phase re-weaving strategy, that used a quick initial re-weaving, and possibly re-weaved again with a more optimized weaver, perhaps upon demand. This would be much like Just-In-Time (JIT) compilers, where further processing is delayed until execution demonstrates its need.

5.1.2 Treating Pointcuts As Predicates

Apostle's pointcuts are deliberately considered as predicates for performance reasons. Being purely functional, the order of evaluation of components of a compound pointcut commute, which means they can be re-ordered by Apostle with no semantic difference. Such a re-ordering may eliminate large swathes of the image from consideration and have a dramatic impact on performance, and thus contribute to maintaining the expected liveness.

This point is illustrated with the following pointcut:

```
reception(#at:) & kindOf(Array)
```

Since *Object*, the root of the Smalltalk class hierarchy, implements *#at:*, then the *reception()* matches every class. Thus using the order defined would require identifying all the possible reception points of *#at:* requires querying all classes of the system, and then verifying whether such classes are a kind of *Array*. This is an expensive operation, as seen in Table 3. Re-ordering the pointcut to first find *Array* and traversing its hierarchy reduces the search space dramatically and hence the execution time, as also shown in Table 3.

5.2 Implications of AOP in Self-Hosted Environments

The Apostle weaver, Smalltalk compiler and the development environment and the applications under development all run in the same virtual machine (VM). This introduces potential for strange side-effects not seen in most other AOP environments.

5.2.1 Vulnerability of the Development Environment

Since Apostle and the compiler co-exist within the same image as applications and the system, a stray pointcut or advice may inadvertently clobber the system. Imagine advice targeting the the compiler which causes a run-time error. This would be impossible to remove as the compiler is necessary to regenerate the shadows. This actually occurred on one occasion during Apostle's development.

These situations will occur in self-hosted environments like Smalltalk. One possible solution might allow an explicit declaration of particular code-bases as being *out of bounds*, such as the development environment. But targeting the classes of the development environment can prove useful. Another solution might be to provide some indication to the programmer of the impact of a pointcut, such as whether the impact is localized to the module, to the whole application, or beyond the application. An icon or warning could be output for a pointcut that targets join points outside of its module.

5.2.2 Side-Effects From Cflow()

Incremental development in a self-hosted AOP development can also reveal previously-unseen changes to program semantics. This is better illustrated with an example, such as the following piece of advice:

```
after<cflow(p1())>
    ...
```

This advice is to be executed while in the context of any join points identified by *p1()*. Now consider redefining *p1()* to:

```
pointcut p1(): kindOf(Process)
```

Process is a Smalltalk thread, any execution is always done by a Process. Thus upon saving this pointcut redefinition, any advice targeting *p1()* will commence being dispatched. Hence any advice targeting this *cflow()* should also begin being dispatched upon definition.

This dispatch will not occur in Apostle due to our implementation of *cflow()*. As described in Section 4.1.5m *cflow()* is itself implemented using *around* advice, such that a process-marker is installed for the duration of any join points matched by the *cflow()* sub-pointcut (i.e. *p1()*). The residual test for the *cflow()* is then a quick test that the process marker has been set. Since the Process code is usually only executed at the beginning of a process' life, the advice will not execute for already running processes.

An alternative implementation for *cflow()*, that would provide the correct semantics, would be to test the entire process stack at runtime. But while more complete, this would impose significant overhead. And while this is an interesting situation, we do not anticipate that it will occur often, as developers rarely modify actively running applications.

5.2.3 Presenting Un-named Structures

Like AspectJ, advice in Apostle is un-named. Un-named advice does not fit well with the Smalltalk environment, where Smalltalk code browsers and compilers generally assume that elements have some identifying tag, such as class or method names. With methods, a new method replaces an old method definition should they have the same name. Lacking such a tag, an advice definition must be assumed to be new advice, and not a modification to existing advice. The programmer must and instead explicitly remove the old advice.

This also causes difficulties to users when attempting to find advice of interest in a browser. Apostle provides some help by incorporating the pointcut name into the advice implementation selector, which happens to be shown by the browsers. But this useful side-effect only helps when using named pointcuts. The programmers must otherwise systematically examine all the advice on the relevant aspect.

This same issue has been experienced with the AJDT support in Eclipse. AJDT provides an advice labelling strategy that incorporates the pointcut definition, and thus encourages people to use named pointcut.

5.3 Future Work

Further evaluation of this work will benefit from optimizing the the incremental re-weaving algorithm presented in Section 4.2. This is a general-purpose algorithm, that can process any number of simultaneous additions and removals. In practice, however, the algorithm is used to handle only single addition or removals: programmers can only change a single program element at a time using the Smalltalk browsers. Weaving performance algorithm could potentially be improved by using a customized algorithm for each particular change. Improving class redefinitions times is a definite requirement.

Another direction of future work is to accommodate method caller side join points, which may be accomplished using type inferencing. Type inferencing could also be used to further eliminate shadows from consideration.

This work may be more broadly applicable to incorporating aspect-oriented technologies to any incremental environment where a program can be dynamically extended after compile-time, such as replacing portions of the program, or by dynamically loading new components. Such environments require additional support for AOP, as these existing program portions may be targeted by the aspect programs. Indeed, portions of the program may also be removed, including AOP constructs, possibly requiring removal of aspect programs. While the Apostle implementation resulting may not be directly transferable, we believe the lessons learned and weaver design may be informative for other efforts.

6. CONCLUSIONS

This paper discussed issues surrounding the implementation of Apostle, an incremental weaver for Smalltalk. It briefly described the language binding, and provided rationales of its differences from AspectJ. It also described the target model, showing how the aspect-oriented elements are transformed and transforms the image.

The paper then presented an incremental weaver for this implementation. Three necessary conditions were required to enable this incremental re-weaving:

1. the dependencies and assumptions made in the model must be well understood;
2. the possible changes to the program must be analyzed to understand how they break the dependencies and assumptions;
3. the implementation must obtain notification of all changes to the program, so as to correct the model.

Apostle's incremental weaver maintains sufficient information, derived from understanding these conditions, to ensure that compilation and re-weaving is done efficiently and does not introduce noticeable latency.

Finally, the paper discussed some of the trade-offs in the implementation, and outlined areas for future work.

Acknowledgments

The authors would like to thank Gail Murphy, Mik Kersten and Annie Ying for their valuable suggestions in improving this paper.

This work was partially supported by grants from the Natural Sciences and Engineering Research Council of Canada, Xerox Corporation of Canada, and Sierra Systems PLC.

7. REFERENCES

[1] The AspectJ homepage. `http://aspectj.org`. Verified 2002/03/15.

[2] K. Böllert. Implementing an aspect weaver in Smalltalk. Position paper at STJA '98, Erfurt, Germany, Oct. 1998.

[3] K. Böllert. *AOP/ST User's Guide*, 1999.

[4] B. de Alwis. Aspects of incremental programming. Master's thesis, University of British Columbia, Vancouver, Canada, 2002.

[5] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Mass., 1984.

[6] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[7] K. Gybels. Aspect-oriented programming using a logic meta programming language to express cross-cutting through a dynamic joinpoint structure. Master's thesis, Vrije Universiteit Brussel, 2001.

[8] R. Hirschfeld. AspectS – aspect-oriented programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *LNCS*, pages 216–232. Springer-Verlag, 2003.

[9] G. Kiczales. Coding the observer in AspectJ. Message to the `aspecj-users@aspectj.org` mailing list, July 21 2001.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, Oct. 2001. Special Issue on Aspect-Oriented Programming.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspect-oriented programming with AspectJ. In *Proc. 15th European Conf. on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2001.

[12] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proc. Compiler Construction (CC2003)*, volume 2622 of *LNCS*, pages 46–60, 2003.

[13] Object Technology International, Inc. *Eclipse Platform Technical Overview (Updated for Eclipse 2.1)*, Feb. 2003.

[14] The Squeak homepage. `http://www.squeak.org`. Verified 2003/03/15.

[15] The Eclipse Project. AspectJ development tools project. `http://www.eclipse.org/technology/`. Verified 2003/09/21.