Jumping Aspects Revisited

Bruno De Fraine, Wim Vanderperren, Davy Suvée
System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
bdefrain,wvdperre,dsuvee@vub.ac.be

Johan Brichau
Programming Technology Lab (PROG)
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
jbrichau@vub.ac.be

ABSTRACT

In this paper, we propose an extension of the JAsCo aspect-oriented programming language for declaratively specifying a protocol fragment pointcut. The proposed pointcut language is equivalent to a finite state machine. Advices are attached to every transition specified in the pointcut protocol. We claim that stateful aspects benefit from run-time weaving and therefore introduce the JAsCo run-time weaver. When employing this run-time weaver, a stateful aspect is only woven at the join points it is currently interested in. When a state-change occurs, it is rewoven to the new target join points. Hence, a real *jumping aspect* is realized, that literally jumps from join point(s) to join point(s).

Keywords

Aspect-Oriented Software Development, Run-Time Weaving, Stateful Aspects, JAsCo

1. INTRODUCTION

An aspect definition generally consists of two separate parts: the aspect applicability condition (pointcut specification) and the aspect functionality (advice). The aspect applicability condition determines when the aspect's functionality should be invoked. In early aspect-oriented languages, this condition was often expressed in terms of static locations in the base program. However, it was argued early on that conditions in terms of run-time events were more appropriate (e.g. jumping aspects [3], AspectJ's dynamic join point model [19], event-based AOP [10],...).

The importance of dynamic applicability conditions is very well illustrated by a relatively new kind of aspects: *stateful aspects* [8]. These aspects describe their applicability in terms of a sequence of run-time events. The true dynamic nature of their applicability condition even implies that these aspects can benefit from a *run-time weaver* that enables aspects to 'jump' in a very literal sense. This concept is illustrated in the context of JAsCo [28], which is a dynamic aspect language with support for the definition of stateful aspects, and which provides a run-time weaver for dynamically adding and removing aspect behavior.

The following section describes how stateful aspects benefit from a run-time weaver. Section 3 describes stateful aspects in JAsCo, together with an analysis of the drawbacks of static weaving. Section 4 describes run-time weaving in JAsCo and weaving of stateful aspects using run-time weaving. Section 5 describes related work and section 6 concludes the paper and discusses future work.

2. STATEFUL ASPECTS BENEFIT FROM RUN-TIME WEAVING

Stateful aspects [8] are aspects that are triggered by the occurrence of a consecutive sequence of events. They are considered stateful because their applicability condition needs to consider the notion of *state* to keep track of the (past) sequence of events. In contrast, the applicability condition of traditional aspects is stateless because the aspect applies on *all* the events that it matches. For example, consider a stateful logging aspect that only requires to write data to a log if a user is logged in. The applicability condition of this stateful logging aspect can be described in terms of a sequence of events. The first event is the execution of the `login()` method. Subsequent events trigger the aspect's advice that writes data to a log until the execution of the `logout()` method, which terminates the applicability of the stateful logging aspect. Another example, which is used in this paper, is a simple publish/subscribe collaboration, implemented using an aspect. In such a stateful publish/subscribe aspect, the aspect only publishes to a subscriber *after* it has actually subscribed, i.e. the publish advice is only applicable after the execution of the `subscribe()` method.

Some AOP languages, such as JAsCo [28], provide linguistic support for the description of stateful aspects. In JAsCo, the aspect applicability condition of stateful aspects is described by means of a declaratively specified *protocol* that describes the desired sequence of events. An implementation of such a protocol in JAsCo is shown later on, in section 3.2. Although stateful aspects could be simulated using traditional aspects by keeping an explicit state variable in the aspect code, such a manual implementation is a cumbersome and error-prone task. More importantly, it involves tangling the aspect-applicability mechanism with the aspect's functionality inside of the advice, which is undesirable. Providing

linguistic support for the implementation of stateful aspects frees the programmer from the burden of writing the state bookkeeping code, which allows for a trivial and clean implementation of stateful aspects.

However, the weaving of stateful aspects using a static weaver results in a significant performance overhead, as code needs to be woven at any possible join point that is considered by the stateful aspect. Although a stateful aspect might be interested in many different join points in its lifetime, only a limited set of join points are applicable at a particular point in time. Hence, the stateful aspect is only interested in the occurrence of a particular subset of events, i.e. those events that are applicable given the aspect's state. Consequently, the presence of woven code at those join points that are not applicable only induces a performance overhead. Static weaving of stateful aspects also results in a severe limitation in expressiveness because the possible join points need to be known at compile-time. Some stateful aspects however require to compute the applicable join points at run-time. For instance, reconsider the case of the simple publish/subscribe protocol. In some cases, a decision about the events that should be published can only be taken upon subscribing (i.e. at run-time). This means that the aspect can only be woven at the required join points after the subscription event.

To resolve these problems, we propose to weave stateful aspects at run-time. Instead of statically weaving code at any possible join point that might be required for the execution of the stateful aspect, the aspect is only woven at the applicable join points at run-time. In essence, when a stateful aspect changes state (because an event expressed in the protocol has occurred), the run-time weaver unweaves the code at those join points in which the aspect is no longer interested and reweaves it at the appropriate join points. This realizes a real *jumping aspect* that literally jumps from join point(s) to join point(s). As a result, no unnecessary woven code is left at inapplicable join points. Furthermore, a run-time weaver provides the opportunity to determine the join points at run-time. Also notice that when the aspect is no longer applicable, it is completely unwoven and thus does not cause a performance overhead any longer. Previous work by Costanza [6] also motivates that aspects should be able to vanish.

Run-time weaving thus provides a natural and efficient technique for the realization of stateful aspects. Of course, run-time weaving itself also causes a significant performance overhead. Nevertheless, for stateful aspects that do not require frequent reweaving (i.e. do not change state very often), the performance gain from the absence of woven code at unnecessary join points can be far greater than the performance loss induced by run-time weaving.

3. JASCO LANGUAGE

3.1 Introduction

The JAsCo [28] AOP approach is an aspect-oriented extension for Java that allows for a clean modularization of crosscutting concerns. The JAsCo language tries to stay as close as possible to the original Java syntax and concepts, and introduces only two new entities, namely *Aspect Beans* and *Connectors*. An aspect bean is an extended version of a regular Java Bean that allows describing crosscutting concerns independently of concrete component types and APIs. JAsCo connectors on the other hand are used for deploying one or more reusable aspect beans within a concrete component context and provides support for describing their mutual interactions.

```
1  class PublishManager {
2
3    // Bookkeeping/notification code
4    void addListener(MethodListener ml) { ... }
5    void notifyListeners(String methodname, Object[] args) { ... }
6
7    hook Publish {
8      Publish(topublish(..args)) {
9        execute(topublish);
10     }
11
12     after() {
13       notifyListeners(thisJoinPoint.getMethodName(), args);
14     }
15   }
16 }
```

Figure 1: A JAsCo aspect bean for simple publish behavior

The JAsCo language is illustrated by implementing the simple publication behavior that was mentioned in the previous section. Suppose the execution of certain methods of a component should be published to interested listeners. It should however be avoided to tangle the components logic with code that manages this publication system. JAsCo allows to specify this behavior as a reusable aspect bean that is illustrated in figure 1. Note that an aspect bean looks very similar to a regular Java Bean, and likewise implements a number of ordinary Java class members. In addition, one or more hook definitions that implement the crosscutting behavior can be specified. For example, the `PublishManager` aspect bean contains a number of standard methods to manage and notify the listeners (lines 3-5), and a `Publish` hook (lines 7-15) that is responsible for invoking the notification after the execution of a relevant method. A hook has one or more constructors that specify in an abstract way *when* the behavior should be triggered, and one or more advice methods (`before`, `around`, `after` ...) that specify *what* this behavior consists of. In this case, the constructor (lines 8-10) outlines that the hook behavior applies to the execution of the abstract method `topublish`. The `after()` advice method (lines 12-14) then specifies that, after this event, the listeners should be notified of the method name and the arguments.

JAsCo's abstract and reusable aspect beans are deployed onto a concrete component context by making use of connectors. Each connector allows to explicitly instantiate and

```
1  static connector PublishUpdates {
2
3    PublishManager.Publish publish =
4      new PublishManager.Publish(void ComponentX.update*(*));
5
6    publish.after();
7
8  }
```

Figure 2: A JAsCo connector for the publishing of updates

initialize one or more logically related hooks. Figure 2 illustrates a connector that instantiates the `Publish` hook of figure 1 onto the update methods of the `ComponentX` component. This is realized by passing these methods as a wildcard to the hook constructor (lines 3-4). A connector additionally allows to select and order the behavior methods of the instantiated hooks. In this case, it is specified that the `after()` advice method of the `Publish` hook should be executed whenever a join point of this hook is encountered (line 6). As a result of the declarations in the connector, the `Publish` hook is applied to the update methods of `ComponentX`, and as such, registered listeners will be notified after the execution of these methods.

Now consider an extension of the `PublishManager` aspect bean where this publication only occurs *after* a certain subscribe method has been executed. To capture this behavior using traditional (stateless) aspect facilities, we have to manually implement code that maintains a state regarding this condition. This approach is illustrated in figure 3, where the extended `ConditionalPublishManager` aspect bean is presented. It contains two hooks: the new `Subscribe` hook is responsible for setting the `subscribed` instance variable to true after the execution of an abstract subscribe method, and the `ConditionalPublish` hook extends the `Publish` hook to make the publishing behavior apply only when the state variable `subscribed` is true. In the specification of these hooks, the usage of the `isApplicable()` hook method is crucial. This JAsCo language construct allows to describe a run-time condition for a hook as the advices are only executed when the body of this method evaluates to true (similar to the `if` pointcut construct in AspectJ). For the first hook, the `isApplicable()` method in line 10 specifies that the hook should not execute when the `subscribed` variable already has a true value. For the second hook, the method in line 18 specifies that the publishing behavior should only apply when the `subscribed` variable reflects a subscribed state. As no other elements of the `Publish` hook are modified, they can be inherited as such.

Although it is possible to implement the desired functionality using only stateless aspect facilities, this is quite a cumbersome and error-prone task, since it requires to capture each state in a separate hook and involves adding code to maintain variables regarding this state. In the next section, a stateful extension to the JAsCo language is presented which solves these problems by allowing the developer to declaratively specify a protocol of expected pointcuts.

```
1  class ConditionalPublishManager extends PublishManager {
2
3    boolean subscribed = false;
4
5    hook Subscribe {
6      Subscribe(subscribe(..args)) {
7        execute(subscribe);
8      }
9
10     isApplicable() { return !subscribed; }
11
12     after() {
13       subscribed = true;
14     }
15   }
16
17   hook ConditionalPublish extends Publish {
18     isApplicable() { return subscribed; }
19   }
20 }
```

Figure 3: An extended JAsCo aspect bean for conditional publish behavior

3.2 Stateful Aspects Language

Mainstream aspect-oriented approaches rarely support protocol history conditions. In many cases, it is only possible to refer to previous join points when they still have an activation record on the stack (i.e. using the `cflow()` keyword in AspectJ). In order to solve this limitation, Douence et al. [8] propose a formal model for aspects with general protocol based triggering conditions, named *stateful aspects*. In this section, we illustrate how the JAsCo language is extended with stateful pointcut expressions, based on this formal model.

```
1  class StatefulPublishManager extends PublishManager {
2
3    hook PublishSubscribe {
4
5      PublishSubscribe(subscribe(..args),
6                       topublish(..args)) {
7
8            Waiting: execute(subscribe) > Publish;
9            Publish: execute(topublish) > Publish;
10     }
11
12     after Publish() {
13       notifyListeners(thisJoinPoint.getMethodName(), args);
14     }
15   }
16 }
```

Figure 4: A JAsCo stateful aspect bean for the simple subscribe/publish protocol

To illustrate the JAsCo stateful aspects syntax, reconsider the simple publish-subscribe protocol from the previous sections. Only when a subscription event occurred, the aspect should start publishing. Figure 4 illustrates how this protocol can be declaratively described by making use of the JAsCo stateful aspect language. The constructor of the stateful hook `PublishSubscribe` (line 5-10) describes a protocol-based pointcut expression. Every line in the constructor defines a new transition within the protocol. Each transition is labeled with a name (e.g. `Waiting`), defines a JAsCo pointcut expression (e.g. `execute(subscribe)`) and specifies one or more destination transitions that are matched after the current transition is fired. A transition fires when its pointcut expression evaluates to true. For example, the `Waiting` transition only fires whenever the concrete method(s) bound to the abstract method parameter `subscribe` are executed. In that case, transition `Publish` is activated and will be evaluated for the subsequent join points encountered during the application's execution.

A stateful aspect always starts by evaluating the first defined transition. As a result, a protocol `subscribe-topublish` is described. In between the fired transitions, other join points can also be encountered. As such, a sequence of events `methodY-subscribe-methodX-topublish` is also a valid instance for the defined protocol and will trigger the associated transitions.

On every transition defined in the stateful constructor, ad-

vices can be attached which are executed whenever the transition is fired. For example, the `after Publish` advice (line 12-14) is only triggered whenever the transition `Publish` is fired. In other words, the advice is executed whenever the concrete method(s) bound to the abstract method parameter `topublish` are executed in that state of the stateful aspect. To sum up, the stateful `PublishSubscribe` hook will only start notifying interested subscribers when a subscription event occurred.

Figure 5 illustrates how the stateful aspect of figure 4 is instantiated and deployed using a JAsCo connector. This example is similar to the connector of figure 2 as it bounds the abstract method `topublish` to the update methods of `ComponentX`. Additionally, the `subscribe` abstract method parameter is bound to a concrete subscription method of a certain `PSComponent`. Consequently, as soon as this subscription method has been executed, the aspect will start intercepting update methods on `ComponentX` and will start notifying its registered listener(s).

```
1 static connector PSConnector {
2   StatefulPublishManager.PublishSubscribe ps =
3     new StatefulPublishManager.PublishSubscribe(
4       boolean PSComponent.subscribe(),
5       void ComponentX.update*(*)
6     );
7 }
```

Figure 5: The JAsCo connector for deploying the stateful `PublishSubscribe` hook.

3.3 Advanced Language Features

In addition to attaching advices on each transition separately, it is also possible to describe global advices that are triggered for all fired transitions. In this case, the advice is specified as usual, but the transition label is omitted. It is also possible to attach a specific `isApplicable` method to a particular transition in the protocol. Hence, the transition will only fire when both the pointcut expression and the `isApplicable` condition evaluate to true. Likewise to advices, a global `isApplicable` condition can be specified which is applied to all transitions. In that case, transitions are only fired when they satisfy their pointcut expression and both the global and local `isApplicable` conditions. The following code fragment shows both a global and local `isApplicable` condition.

```
1  isApplicable() {
2   // global condition for all transitions
3  }
4  isApplicable XTrans() {
5    // local condition only relevant for the transition XTrans
6  }
```

The JAsCo stateful aspects constructor can also specify multiple destination transitions for a given transition. The syntax is illustrated in the code fragment below. After firing the `XTrans` transition, both the `YTrans` and `QTrans` transitions are evaluated for subsequent encountered join points (line 4). Note that the destination transitions are evaluated in the sequence defined in the destination expression. As such, when both the `YTrans` and `QTrans` transitions are applicable for a given join point, only the `YTrans` transition will be fired and only the `YTrans` destination transitions will be evaluated for subsequent encountered join points. This allows to keep the protocol deterministic and efficient to execute. It is also possible to omit a destination transition for a certain transition. In that case, when the transition fires, no more transitions need to be evaluated and the aspect *vanishes*. This concept is illustrated by the `QTrans` transition (line 6). Also notice that this transition describes a more involved pointcut designator using the `cflow` keyword.

In case the stateful aspect requires to start by evaluating more than one transition, the `start` keyword can be employed. This keyword is followed by a list of starting transitions for matching join points when the aspect is deployed. Multiple start transitions are specified similarly to multiple destination transitions, by using `||` as delimiters. When no start transition is specified, the first defined transition is used as the starting one.

```
1  //starting with two transitions:
2  start > XTrans || QTrans;
3  //two destination transitions:
4  XTrans: execute(methodA) > YTrans || QTrans;
5  //no destination transition:
6  QTrans: execute(methodB) && !cflow(methodC);
7  YTrans: execute(methodC) > YTrans;
```

The syntax proposed in the previous paragraphs provides a way for specifying powerful protocols but might be too tedious in case of simple protocols. Therefore JAsCo also supports a simpler syntax for protocols that do not require multiple destination transitions for a given transition. The following code fragment illustrates a constructor that is equivalent to the constructor of figure 4. Labeling transitions is still possible in order to be able to attach local advices to specific transitions.

```
1  PublishSubscribe(subscribe(..args), topublish(..args)) {
2    execute(subscribe) > Publish: execute(topublish) > Publish;
3  }
```

The JAsCo stateful aspect language also supports triggering aspects on the opposite (complement) of a protocol. Furthermore, JAsCo stateful aspects are non-strict per default, i.e. they allow non-specified intermediate transitions. Specifying strict protocols is also supported. The discussion of these features is however outside of the scope of this paper. The interested reader is referred to [30, 15] for more information.

3.4 Implementing Stateful Aspects

A naive approach to realize a stateful aspect would be to weave it at all possible join points defined within its protocol. This induces a performance overhead at all these join points, while the stateful aspect is only interested in a limited set of join points corresponding to the subsequent transitions that are to be evaluated. In order to implement stateful aspects more efficiently, a genuine run-time weaver is required which is able to reweave the stateful aspect each time a transition is fired.

Another major problem with statically weaving stateful aspects is that the pointcuts have to be defined in advance.

As argued in section 2, it would be interesting to dynamically decide the concrete join points that have to be used for triggering the subsequent transitions. In that case, a static binding of the abstract method parameters of the hook constructor is not possible and a run-time weaver is necessary for reweaving the stateful aspect after a transition has fired.

To address these shortcomings, we propose to employ a run-time weaver. The following sections introduce the JAsCo run-time weaver and explain how stateful aspects are implemented employing this weaver.

4. TOWARDS RUN-TIME WEAVING

The JAsCo technology was originally *trap*-based. At every join point a trap is inserted that defers execution to the JAsCo run-time infrastructure. This infrastructure will trigger any aspects that apply to the join point or, when no aspects are applicable, it will return to the normal execution. As such, dynamic aspect addition and removal becomes possible because the aspects behavior itself is not statically woven. The JAsCo distribution contains a preprocessor tool that inserts traps at all possible join points before run-time by transforming the necessary classes. These transformations are performed through the byte-code adaptation library Javassist [5]. The main problem of this preprocessing approach is performance. Applications equipped with JAsCo traps execute often more than ten to twenty times slower than the original, which is unacceptable. This overhead stems in part from the naive interception system, namely inserting traps at all possible join points.

In order to improve the performance of JAsCo, the JAsCo HotSwap framework [29] is introduced. HotSwap is a custom-made byte-code instrumentation framework that allows altering the byte-code of a class, even if it is already loaded into the virtual machine. As such, it is possible to install traps just-in-time when a new aspect is added to the system. Likewise, the original method byte-code is re-installed when the aspect is removed and no other aspect are applicable. JAsCo HotSwap has two different implementations, depending on the virtual machine version. For Java 1.4, HotSwap employs the Java Debugging Interface (JDI) to dynamically replace classes. When a 1.5 compatible virtual machine is detected, HotSwap employs the novel Java Programming Language Instrumentation Services (JPLIS) API, which avoids running the virtual machine in debugging mode. Both libraries are standard libraries available in most standalone (i.e. not embedded) virtual machine implementations, making JAsCo perfectly portable over a wide range of platforms. They also make sure that the application is left in a consistent state after byte-code of a class has been replaced. The byte-code manipulations themselves are also performed through the Javassist library.

In principle, JAsCo HotSwap already suffices to efficiently implement stateful aspects as it allows to only insert traps to join points where the stateful aspect is currently interested in. However, by inserting traps that refer to the JAsCo run-time infrastructure, the performance of JAsCo is still not optimal. In several benchmark experiments, the JAsCo advice execution performance is measured [29, 15] to be five to ten times slower than the statically woven language AspectJ. This overhead is mainly caused by the additional indirection these traps impose. In addition, the traps have a fixed implementation for every possible advice attached, so they have to capture all possible relevant run-time information. However, capturing the actual arguments in an array for instance, is a very expensive operation. When the actual arguments are not required in the advices, a substantial performance gain can be realized by avoiding capturing this run-time information.

4.1 The JAsCo Run-Time Weaver

In order to improve the run-time performance of JAsCo AOP, a run-time weaver is proposed. Instead of inserting traps, a highly optimized code fragment is inserted into the target join points. This code fragment directly invokes all applicable advices in the correct sequence and thus avoids the indirection through the JAsCo run-time infrastructure. The JAsCo approach is however a dynamic AOP approach. As such, the woven join point behavior might become invalid. This event occurs when a connector is added that instantiates a hook that is applicable on a join point where aspects are already attached or when a connector is removed that contains an applicable hook for such a join point. 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 JAsCo run-time weaver is able to cope with these issues. When no advices are applicable any longer, the original byte-code of the method is reinstalled.

Generating optimized code for a target join point is not always achievable because some pointcut expressions have to be re-evaluated for every execution of the join point (precisely because of the dynamic join point model). For example, when a hook defines a `cflow` condition in its constructor, this constructor has to be re-evaluated for every execution of a join point. The entire constructor body does not have to be re-evaluated however. In this case, only the result of the `cflow` condition is able to change for different executions of the join point. As such, partial evaluation techniques are 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 [26].

Another major optimization of the JAsCo run-time weaver consists of detecting which static and dynamic reflective join point information the aspects might require. Suppose for instance that an aspect only requires the method name of the current join point. In that case, most AOP implementations still capture all actual arguments in an array, which is a very expensive operation, even though they are not required. The JAsCo aspect bean compiler analyzes in detail which contextual join point information is required and stores this information in the compiled aspect so that the run-time weaver can exploit this. If the applicable aspects at a join point only require the method name for instance, only this information is captured. Obviously, because this detection happens at compile-time, it has to be conservative and thus might still capture too much. For example, if a logging advice contains a dynamic test for selecting whether it logs only the method name or also the arguments, the advice is analyzed to require the actual arguments and the method name, while in some cases it only requires the method name. Nevertheless,

this analysis allows for a significant optimization in a large number of cases.

The main drawback of the run-time weaver is the increased run-time overhead for adding and removing aspects. In the trapped approach, when a trap is already placed, adding a new aspect does not require any HotSwap overhead whatsoever. Also, even if a new trap has to be inserted, this is a lot less costly than weaving because the code for the trap itself remains constant whereas with run-time weaving, a new code fragment has to be computed for each individual join point. In order to address this overhead, JAsCo is still able to combine the regular preprocessing approach with the run-time weaver and even with the trapped HotSwap approach. Classes that are preprocessed to include traps are never subject to run-time weaving. In addition, it is possible to define a global function that dynamically decides whether a trap is inserted or whether the run-time weaver is employed. This function has the following signature:

```
boolean inlineCompile(JoinPoint jp, Vector hooks)
```

When the method returns true, the run-time weaver is employed, otherwise a trap is inserted. Both reflective information about the join point and the list of applicable hooks are available for deciding whether run-time weaving is appropriate. As such, a heuristic function can be implemented that for example only activates the run-time weaver for join points that are executed more than twenty times in the past second. JAsCo thus effectively combines and integrates three alternative aspect weavers.

4.2 Performance Evaluation

In order to evaluate the performance of the novel JAsCo run-time weaver, we employ the AWBench [17] benchmark[1]. This benchmark is a project of the AspectWerkz team and is especially created to compare the performance of dynamic AOP systems. AWBench is a micro benchmark and consists of 12 tests, all advising a single method in a different way. Every test is executed two million times and the average execution time of the method is recorded. When a certain test is not directly supported by the AOP approach, it is simulated using the best available alternative (e.g. when no *after throwing* advice is available, it is simulated using around advice). We compared the performance of JAsCo with the following AOP approaches: AspectJ 1.2 [19], JBoss/AOP 1.0 [16], AspectWerkz 2.0 [18], Spring/AOP 1.1.1 [27], dynaop 1.0 Beta [11] and cglib 2.0.2 [4]. The next paragraph shortly introduces the technologies employed in each of these approaches. Notice that this selection is not meant as a comprehensive overview of dynamic AOP approaches. Nevertheless it includes a significant portion of the practically used dynamic AOP systems.

AspectJ and AspectWerkz both use a traditional weaver that invasively weaves the aspects into the target classes at run-time. Similar to JAsCo, AspectWerkz also features a genuine run-time weaver while AspectJ is limited to compile-time weaving. JBoss/AOP uses an approach similar to the original JAsCo technology, namely inserting traps to all advised join points. In contrary to JAsCo, the traps are installed at load-time and can never be removed. As such, at join points where no traps are attached, dynamic aspect interference is impossible. Spring/AOP and dynaop are two proxy-based approaches that employ the Java Dynamic Proxies feature to dynamically attach advices to objects. Dynamic Proxies are instance-based, so it is easily possible to only advice one object of a certain class. The main drawback however is that class-based aspect application is more difficult to realize. In addition, Dynamic Proxies induce a relatively high performance overhead. cglib (Code Generation Library) is not an AOP framework per se, but a byte-code adaptation library with extensive AOP features.

Figure 6 on page 7 illustrates the results of running the AWBench with the introduced approaches. The performance of JAsCo using the trapped approach to attach aspects is also measured. Notice the logarithmic scale of the results to fit all results in one clear chart. In all benchmarks the three approaches that use weaving (JAsCo, AspectJ, AspectWerkz) perform significantly better than the others. In the most simple before advice for example, JAsCo executes more than a hundred times faster than Spring/AOP. The trapped approaches (JBoss/AOP and JAsCo no-RTW) perform worse than weaving but still execute considerably faster than the proxy-based approaches (Spring/AOP and dynaop).

For the before advices where the run-time context is fetched declaratively, the three woven approaches perform equally well. All three optimize the join point interception to only fetch that data that is requested. When reflection is used however, JAsCo is able to improve on both AspectWerkz and AspectJ. This is because JAsCo has a fine-grained required context detection, also when it is reflectively queried. `thisJoinPoint` vs. `thisJoinPointStaticPart` is the only difference accounted for by AspectJ and AspectWerkz. When `thisJoinPoint` is employed, all possible run-time context information (target object and type, caller object and type, actual arguments and types, etc...) is stored, whereas only a fraction of this dynamic information might be effectively required.

In addition, when several advices are combined or when an around advice is employed JAsCo seems to improve more significantly on AspectJ and certainly on AspectWerkz. In all the other tests JAsCo, AspectJ and AspectWerkz are very close. As such, it seems that the performance of compile-time and run-time weaving approaches converges and probably a boundary of traditional weaving has been reached. In any case, the run-time performance of JAsCo has been improved quite considerably when comparing it with the trapped approach and thus the goal of the run-time weaver has been accomplished.

4.3 Implementing Stateful Aspects using the Run-Time Weaver

The previous paragraphs motivate and explain the JAsCo run-time weaver. As explained before, stateful aspects also particularly benefit from run-time weaving. A naive approach for integrating a stateful aspect would be weaving it at all possible join points defined within the protocol. This induces a performance overhead at all these join points, while the stateful aspect is only interested in a limited set of

[1]The AWBench distribution including JAsCo can be downloaded here: http://ssel.vub.ac.be/jasco/awbench

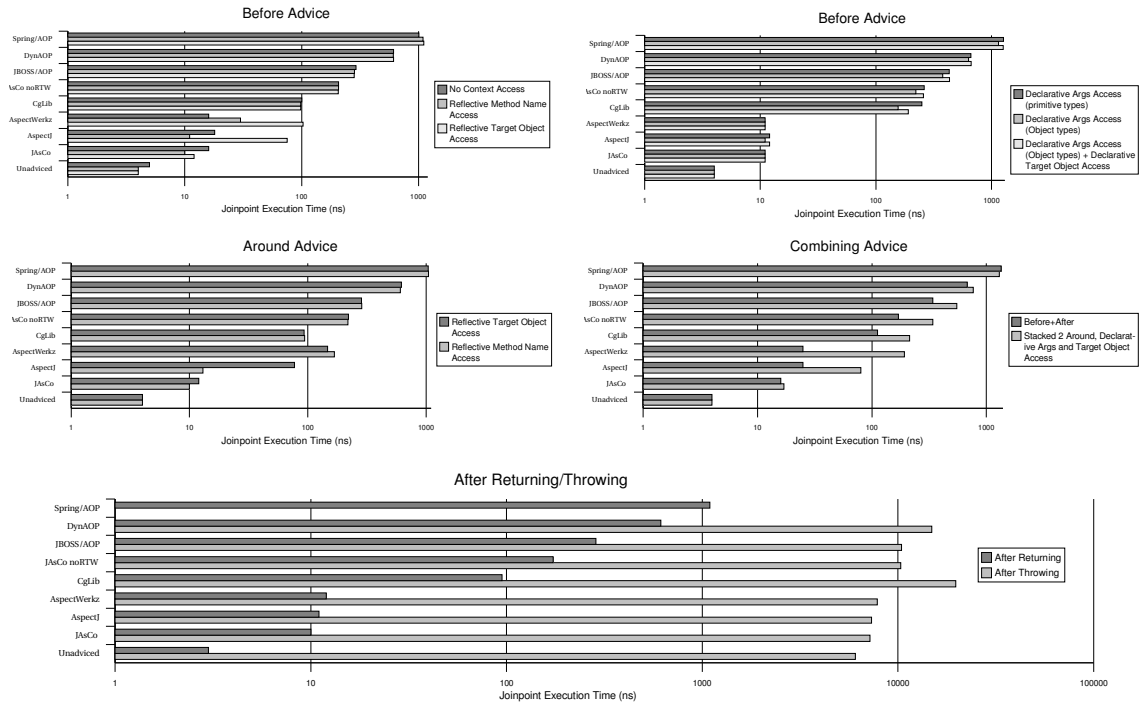

Figure 6: AWBench benchmark results run on a PENTIUM4, 2GHZ, 256 RAM with Ubuntu Linux 4.10, Java 1.5.0 update 1. Notice the logarithmic scale of the timings in order to keep every result readable.

join points corresponding to the subsequent transitions that are to be evaluated. By employing the run-time weaver, it is possible to only weave the stateful aspect at those join points where the aspect is currently interested in. When a transition is fired, the weaver unweaves the aspect at the join points associated with the current transition and weaves it back in at the join points relevant for the subsequent transitions. As such, a real *jumping aspect* is realized. Notice that when the aspect *vanishes* because no subsequent transitions are defined, it is completely unwoven. As a result, no performance overhead for the aspect is endured any longer.

The weaving process itself does however also require a significant overhead. Therefore, when a given protocol is encountered many times in a short time interval, it might be more efficient to weave the aspect at all possible join points of the protocol instead of weaving and unweaving it on-the-fly. This can be configured in JAsCo by using the novel Java 1.5 annotations (meta-data). When the `@WeaveAll` annotation is supplied to the hook, as illustrated by the code fragment below, the run-time weaver weaves the aspect at all join points and never unweaves it unless the aspect itself is manually removed or vanishes.

```
1  @jasco.runtime.aspect.WeaveAll
2  hook StatefulHook { ...
```

In order to implement the stateful pointcut itself, the pointcut is translated to a Deterministic Final Automaton (DFA) [13]. The JAsCo stateful aspects language is equivalent to a DFA because every expression defines one DFA transition, two DFA states and possibly several connection DFA transitions for the destinations. Therefore, the JAsCo compiler compiles a stateful aspect constructor to a DFA that is interpreted at run-time. Every transition of a DFA contains a representation of the pointcut definition and possibly an `isApplicable` condition. When a join point is encountered, the outgoing transitions of the current state are evaluated with the given join point and when a match is encountered, the state machine moves to the destination state. When this event occurs, all associated advices are executed and the aspect is rewoven to the new join points corresponding to the outgoing transitions of the destination state. Because of this implementation strategy, a stateful aspect can be executed very efficiently. It suffices to check only the transitions of the current state, as JAsCo stateful aspect protocols are regular and can be interpreted by a regular DFA. When non-regular protocols are allowed, a history of all relevant encountered events should be maintained, which is very expensive.

5. RELATED WORK

The emerging stateful aspect research is still quite young and at the moment not many AOP approaches support its ideas and concepts. Douence et al. are the first ones to propose an extension of their formal aspect model [7], to support stateful aspects [8]. The advantage of having a formal model is that it allows to automatically deduce possible malicious interactions among aspects. Furthermore, the model supports the composition of stateful aspects using well-defined composition operators. A proof of concept implementation of this model, based on static program transformations, is available [9]. JAsCo improves upon this implementation, as only a subset of join points needs to be woven, whereas a static approach requires to weave all possible join points defined within the protocol.

Walker et al. introduce *declarative event patterns* (DEPs) [31] as a means to specify protocols as patterns of multiple events. Here, AspectJ aspects are augmented with special DEP constructs that can be advised. Their approach is based on context-free grammars, and involves a transformation of the DEP constructs into regular AspectJ aspects that contain an event parser. While DEPs can recognize properly nested events and thus possess an even higher degree of declarative expressibility than the JAsCo approach, they only provide the ability to attach advice code to the entire protocol. Separate transitions of the protocol can as such not be advised, and several overlapping protocols are required to mimic JAsCo stateful aspect behavior. Furthermore, the fact that DEPs lose their identity in a preprocessing step that reduces them to standard aspects, rules out the possibility for optimizations by a weaver that analyzes the feasible transitions of the protocol.

Finally, Masuhara et al. [20] propose an extension of the AspectJ pointcut language to identify join points which are based on the data flow of values within an application. Their novel *dflow* pointcut designator allows to declaratively specify that a particular join point can only match if its arguments are originating from the arguments/return value of a previously encountered join point. By explicitly declaring this preferred data flow, this mechanism allows specifying a more precise pointcut then possible using the current AspectJ pointcut designator language. Although a data flow aspect is not completely similar to a stateful aspect, this research illustrates the need for a mechanism that allows the specification of aspect behavior defined in terms of the history of previously encountered join points. It should however be mentioned that JAsCo stateful aspect are also able to capture data flow pointcuts. This however requires a programmatic approach, which is not as declarative as the approach proposed by Masuhara et al.

Apart from the dynamic AOP technologies employed during our performance evaluation, several other AOP approaches have been introduced for enabling dynamic AOP. Many of these approaches make use of traps and a corresponding registry infrastructure for dynamically (un)weaving aspects. Event-based aspect oriented programming (EAOP) for instance, allows specifying crosscutting concerns by employing event patterns which are described using a formal language [10]. On the implementation level, EAOP inserts traps that query a central execution monitor that has a global view of the executing application and which contains all active EAOP artifacts. In contrast to JAsCo however, EAOP inserts these traps by employing source-code transformations, which obstruct performance optimizations. JAC [21] also make use of traps. Here, these traps are automatically inserted at load-time of the application by making use of byte-code transformations. The Dynamic Aspect-Oriented Platform (DAOP) [22] is an approach that targets legacy component-based systems and allows flexible application of aspects at run-time. DAOP introduces a distributed platform, where a middleware layer is employed for storing the aspect composition information. DAOP does not require any component adaptation and allows aspects to remain first-class entities at run-time.

PROSE [24] and Wool [25] both employ the Java Virtual Machine Debugging Interface (JVMDI) for intercepting the program's execution. A dedicated execution monitor is deployed on top of the JVMDI, which allows capturing the relevant execution events. Whenever an event is encountered where an aspect is applied upon, the corresponding aspect behavior is triggered. Wool improves upon PROSE, as it also allows to invasively insert join points. In addition, aspects are able to implement their own heuristics for deciding whether they should be invasively inserted or not. The Wool heuristics improve on JAsCo as they can be customized on a per-aspect basis whereas in JAsCo only one global heuristics function can be specified.

PROSE2 [23] and Steamloom [2] both aim at achieving an aspect-aware Java Virtual Machine in order to boost the run-time performance of AOP. PROSE2 proposes a next-generation implementation for the original PROSE approach, this by incorporating the execution monitor for joint points into the virtual machine itself. This execution monitor is then responsible for notifying the AOP engine which executes the corresponding advices. Steamloom is implemented as an extension of IBM's Jikes Research Virtual Machine [14] and employs its adaptive optimization system for decorating the base application with aspects. Similar to JAsCo, this mechanism allows for the structure-preserving compilation of reusable aspects which are explicitly deployed at run-time. The main advantage over a run-time weaver is that it avoids the weaving overhead, which makes it very suitable when aspects are deployed and removed frequently.

Filman [12] finally 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.

6. CONCLUSIONS AND FUTURE WORK

This paper presents an extension of the JAsCo language that allows to declaratively specify a regular protocol fragment pointcut. Advices can be attached to each transition in the protocol. Furthermore, we present the JAsCo run-time weaver, of which the run-time performance is able to compete with current state-of-the-art AOP. In this paper we claim that stateful aspects benefit from run-time weaving because 1) they can be executed faster and 2) dynamic pointcut introduction becomes possible. By implementing stateful aspects using the JAsCo run-time weaver, genuine jumping aspects are realized that jump from join point(s) to join point(s) depending on the state of the aspect. It is even possible that the aspect vanishes when no new join points are defined for a certain state.

A limitation of the current stateful aspects language is that it only supports regular protocols. Protocols that require a non-regular language (like for example `n times A; B; n times C`, where `n` can be a different number in every occurance of the protocol), cannot be represented. For instance, in order to enhance the example of figure 4 with an unsubscribe transition so that the aspect is unwoven when

no subscribers are present, a non-regular protocol has to be used because the aspect has to wait for an equal amount of unsubscriptions as subscriptions before it can be unwoven. The advantage of keeping the protocols regular is that they can be efficiently evaluated using a DFA. A naive implementation of a non-regular protocol would require to keep the complete history of all encountered join points in memory, which is not very practical. In literature, several domain-specific optimization techniques for interpreting non-regular languages have been proposed [1]. Extending the JAsCo stateful aspects language to non-regular protocols while still allowing an efficient implementation is subject for future work.

Another intesting area for future work consists of developing heuristics for deciding whether the run-time weaver has to be used or whether the trapped approach is desired. As such, the performance of JAsCo-enabled applications can be automatically tweaked. For long running applications, it could be even possible to exploit learning strategies in order to learn the most optimal heuristic.

7. ACKNOWLEDGEMENTS

Bruno De Fraine and Davy Suvée are supported by a doctoral scholarship from the Institute for the promotion of Innovation by Science and Technology in Flanders in the Industry (IWT).

8. REFERENCES

[1] J. Aycock and N. Horspool. Schrodinger's token. *Software Practice and Experience*, 31(8), 2001.

[2] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proceedings of AOSD*, Lancaster, UK, Mar. 2004.

[3] J. Brichau, W. D. Meuter, and K. D. Volder. Jumping aspects. In *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.

[4] cglib. *cglib Project*. http://cglib.sourceforge.net/.

[5] S. Chiba. Load-time structural reflection in Java. In *Proceedings of ECOOP, LNCS*, Cannes, France, July 2000.

[6] P. Costanza. Vanishing aspects. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

[7] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of GPCE*, Pittsburgh, USA, Oct. 2002.

[8] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of AOSD*, Lancaster, UK, Mar. 2004.

[9] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. *Aspect-Oriented Software Development*, Sept. 2004.

[10] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of REFLECTION*, Kyoto, Japan, Sept. 2001.

[11] dynaop. *dynaop Project*. http://dynaop.dev.java.net/.

[12] R. Filman. Applying aspect-oriented programming to intelligent systems. In *Position paper at the ECOOP 2000 workshop on Aspects and Dimensions of Concerns*, Cannes, France, June 2000.

[13] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory*. Addison Wesley, 2st edition, 2001.

[14] IBM. *The Jikes Research Virtual Machine*. http://www-124.ibm.com/developerworks/oss/jikesrvm.

[15] JAsCo. *JAsCo Distribution Website*. http://ssel.vub.ac.be/jasco.

[16] JBoss Inc. *JBoss/AOP Project*. http://www.jboss.org/developers/projects/jboss/aop.

[17] Jonas Bonér and Alexandre Vasseur. *AspectWerkz AWBench Project*. http://docs.codehaus.org/display/AW/AOP+Benchmark.

[18] Jonas Bonér and Alexandre Vasseur. *AspectWerkz Project*. http://aspectwerkz.codehaus.org/.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP*, Budapest, Hungary, June 2001.

[20] H. Masuhara and K. Kawauchi. Dataflow pointcuts in aspect-oriented programming. In *Proceedings of APLAS*, Bejing, China, Nov. 2003.

[21] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of REFLECTION*, Kyoto, Japan, Sept. 2001.

[22] M. Pinto, L. Fuentes, M. Fayad, and J. Troya. Separation of coordination in a dynamic aspect oriented framework. In *Proceedings of AOSD*, Enschede, The Netherlands, Apr. 2002.

[23] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *Proceedings of AOSD*, Boston, USA, Mar. 2003.

[24] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of AOSD*, Enschede, The Netherlands, Apr. 2002.

[25] Y. Sato, S. Chiba, and T. Michiaki. A selective, just-in-time aspect weaver. In *Proceedings of GPCE*, Erfurt, Germany, Sept. 2003.

[26] D. Serini and O. D. Moor. Static analysis of aspects. In *Proceedings of AOSD*, Boston, USA, Mar. 2003.

[27] Spring. *Spring/AOP Project*. http://www.springframework.org/.

[28] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component-based software development. In *Proceedings of AOSD*, Boston, USA, Mar. 2003.

[29] W. Vanderperren and D. Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In *Proceedings of Dynamic Aspects Workshop*, Lancaster, UK, Mar. 2004.

[30] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful aspects in JAsCo. In *Submitted to SC 2005, LNCS*, Edinburgh, Scotland, Apr. 2005.

[31] R. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, USA, Nov. 2004.