

Jumping Aspects Revisited

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

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

ABSTRACT

Aspects that trigger on a sequence of joinpoints instead of on a single joinpoint are not explicitly supported in current Aspect-Oriented approaches. Explicit protocols are however frequently employed in Component-Based Software Development and business processes and are as such valid targets for aspect application. In this paper, we propose an extension of the JAsCo aspect-oriented programming language for declaratively specifying a protocol fragment. The proposed pointcut language is equivalent to a finite state machine. Advices can be attached to every transition specified in the pointcut protocol. Furthermore, the complement of a protocol can also be used for triggering aspects. The JAsCo tools support the stateful aspects language and implement it very efficiently by employing the JAsCo run-time weaver. As a validation of the approach, we present a case study in the context of reaction business rules.

Keywords

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

1. INTRODUCTION

TODO: focus more on dynamic stuff, dynamicness of stateful aspects...

Aspect-Oriented Programming (AOP) [9] is a recent software programming paradigm that aims at providing a better separation of concerns. At its root is the observation that some concerns cannot be cleanly modularized using traditional abstraction mechanisms such as class hierarchies. These so-called *crosscutting* concerns will therefore inevitably appear scattered across different modules of the system, making them difficult to comprehend and maintain. Typical examples of such concerns are tracing, synchronization and transaction management.

In order to enable a clean modularization of crosscutting concerns, AOP techniques such as AspectJ [10] introduce the concept of an *aspect*, in addition to the use of regular classes. An aspect defines a set of *joinpoints* in the target application where *advices* alter the regular execution. The set of joinpoints is declaratively specified through a *pointcut*. The aspect logic is then automatically *woven* into the target application.

Although AOP research originally focused on a model where aspects are invoked on static locations in the compile-time structure of the program, it was early on argued that the applicability of certain so-called *jumping* aspects [2] can only be expressed in terms of dynamic conditions. Most of the current approaches therefore feature a dynamic joinpoint model, i.e. a model where the joinpoints are run-time events of the program execution. As such, it becomes possible to invoke aspect behavior based on run-time types, call-stack context (e.g. AspectJ's `cflow()` pointcut), dynamically evaluated expressions,...

Describing the applicability of aspects in terms of a *sequence* or *protocol* of run-time events however, is generally not supported. With the exception of the `cflow()` pointcut, the pointcuts of current mainstream AOP languages cannot refer to the history of previously matched pointcuts in their specification. In order to trigger an aspect on a protocol sequence of joinpoints, one is obliged to program code for maintaining a state regarding the occurrence of relevant joinpoints, as such implementing the protocol by hand. Not only is this a cumbersome task, but it is also conceptually undesirable, because it involves mixing the aspect-applicability control-mechanism with the advice code itself.

Explicit protocols are nevertheless frequently encountered in a wide range of fields such as Component-Based Software Development [14, 7], data communications [12] and business processes [6]. We therefore believe that protocols are valid targets for aspect application, and argue that it is desirable to support them in the pointcut language itself; delegating the actual control-mechanism implementation to the weaver. This paper proposes an extension of the JAsCo [11] programming language for *stateful aspects* that can declaratively specify a protocol of expected pointcuts.

The paper is structured as follows. Section ?? introduces the JAsCo language and illustrates the need for explicit support

of protocols. The JAsCo extension for supporting stateful aspects is discussed in section 3. Section 5 focuses on the implementation details of our approach, while section ?? validates it by presenting a case study. Finally, we discuss related work in section 6 and end up with conclusions in section 7.

2. JASCO LANGUAGE INTRODUCTION

TODO: shortly introduce aspect beans, hooks connectors.

3. STATEFUL ASPECTS LANGUAGE

Mainstream aspect-oriented approaches rarely support protocol history conditions. In many cases, it is only possible to refer to previous joinpoints when they still have an activation record on the stack (i.e. using the `cflow()` keyword in AspectJ). In order solve this limitation, Douence et al. [4] 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 ProtocolDynamicTimer extends DynamicTimer {
2
3     hook StatefulProtocolTimer {
4
5         long timestamp;
6
7         StatefulProtocolTimer(methodA(..args),methodB(..args),
8             methodC(..args)) {
9             ATrans: execute(methodA) > BTrans;
10            BTrans: execute(methodB) > CTrans;
11            CTrans: execute(methodC) > ATrans;
12        }
13
14        before ATrans() {
15            timestamp=System.currentTimeMillis();
16        }
17        after CTrans() {
18            long resultingtime = System.currentTimeMillis();
19            notifyListeners(calledmethod,resultingtime-timestamp);
20        }
21
22    }
23 }

```

Figure 1: The JAsCo stateful aspect for dynamically checking a timing contract of a component protocol

In figure ??, an ad-hoc solution was presented for implementing time contract verification of a protocol `methodA-methodB-methodC`. Figure 1 illustrates how the same protocol can be declaratively described by making use of the JAsCo stateful aspect language. The constructor of the hook `StatefulProtocolTimer` (line 7-11) 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. `ATrans`), defines a JAsCo compatible pointcut expression (e.g. `execute(methodA)`) 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 `ATrans` transition only fires whenever the concrete method(s) bound to the abstract method parameter `methodA` are executed. In that case, transition `BTrans` is activated and will be evaluated for the subsequent joinpoints encountered in the application.

A stateful aspect always starts by evaluating the first defined

transition. As a result, a protocol `methodA-methodB-methodC` is described. In between the fired transitions, other joinpoints can also be encountered. As such, a sequence of events `methodA-methodX-methodC-methodB-methodC` is also a valid instance for the defined protocol and will trigger the associated transitions. Notice that the JAsCo stateful aspect pointcut does not have to specify the full protocol of the application; a protocol fragment is sufficient.

On every transition defined in the stateful constructor, advices can be attached which are executed whenever the transition is fired. For example, the `before ATrans` advice (line 13-15) is only triggered whenever the transition `ATrans` is fired. In other words, the advice is executed whenever the concrete method(s) bound to the abstract method parameter `methodA` are executed in that state of the stateful aspect. To sum up, the `StatefulProtocolTimer` hook will take a timestamp before the protocol `methodA-methodB-methodC` is executed and will notify all interested listeners after the full protocol is performed.

3.1 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. As such, the transition will only be fired 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     // returns true or false when advices need to be executed
4 }
5 isApplicable XTrans() {
6     // local condition only relevant for the transition XTrans
7     // returns true or false when advices need to be executed
8 }

```

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 joinpoints (line 2). 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 joinpoint, only the `YTrans` transition will be fired and only the `YTrans` destination transitions will be evaluated for subsequent encountered joinpoints. 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 *dies*. This concept is illustrated by the `QTrans` transition (line 3). 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 joinpoints 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 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 shows a constructor that is equivalent to the constructor of figure 1. Labeling transitions is still possible in order to be able to attach local advices to specific transitions. Notice that the label `start` automatically refers to the first transition.

```

1 StatefulProtocolTimer(mA(..args),mB(..args),mC(..args)) {
2   ATrans: execute(mA) > execute(mB) > CTrans: execute(mC) > start;
3 }

```

Normally, aspects are instantiated explicitly in a connector and this instance is used for all encountered joinpoints. In case of protocol checking stateful aspects, it is sometimes desirable to have a unique instance of the stateful aspect for every execution thread in the application as every thread is typically related to a different interaction. JAsCo allows automatically instantiating multiple instances for a single hook instantiation expression by using specialized keywords in front of the instantiation expression in the connector. The following keywords are supported: `perobject`, `perclass`, `permethod`, `perall`, `percflow` and `perthread`. Thus, in order to obtain a unique aspect instance per execution thread, the `perthread` keyword can be used. The JAsCo run-time system will automatically manage the aspect instances for every thread. This is illustrated by the following code fragment:

```

1 static connector PerThreadConnector {
2   perthread ProtocolDynamicTimer.StatefulProtocolTimer timer =
3     new ProtocolDynamicTimer.StatefulProtocolTimer(
4       void ComponentX.a(), void ComponentX.b(),
5       void ComponentX.c());
6 }

```

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 [sc 2005, JAsCo WS] for more information.

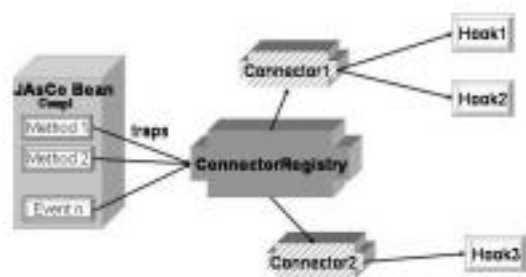


Figure 2: JAsCo run-time architecture.

4. TOWARDS A GENUINE RUN-TIME WEAVER

A naive approach for integrating the stateful aspect would be weaving it at all possible joinpoints defined within the protocol. This induces a performance overhead at all these joinpoints, while the stateful aspect is only interested in a limited set of joinpoints corresponding to the subsequent transitions that are to be evaluated. In order to implement stateful aspects efficiently, a genuine run-time weaver is required which is able to reweave the stateful aspect each time a transition has fired. Therefore, we extend the current JAsCo technology with a run-time weaver. The next section first introduces the current JAsCo technology and afterwards the run-time weaver is presented and evaluated.

4.1 JAsCo Technology Introduction

The JAsCo technology is based on a new component model where traps that enable aspect interaction are already built-in. Ideally, components are shipped employing this new component model. This way, attaching and removing aspects to components implemented in this new component model does not require any adaptation whatsoever to the target beans. Of course, expecting all components to be developed using this new component model is a rather utopian idea. Therefore, it is also possible to automatically transform a regular Java bean into a JAsCo bean by employing a preprocessor that inserts the traps using the byte-code adaptation library Javassist [TODO]. Each trap refers to the JAsCo run-time infrastructure that manages the registered connectors and aspect beans. Figure 2 illustrates this run-time infrastructure schematically. The central connector registry serves as the main addressing point for all JAsCo entities and contains a registry of connectors and instantiated hooks. Whenever a connector is loaded or removed from the system at run-time, the connector registry is notified and its database of registered connectors and hooks is updated dynamically. The left-hand side of Figure 2 shows the JAsCo bean `comp1`. All methods of `comp1` are equipped with traps. As a result, whenever a method is executed, its execution is deferred to the connector registry. The main method of communication of Java Beans is event posting, so firing an event also reschedules execution to the connector registry. When a trap is reached, the connector registry looks up all connectors that registered for that particular method or event. The connector on its turn dispatches to the hooks that have been instantiated with the corresponding method or event.

Our approach is very flexible to support unanticipated run-time changes. Connectors can be easily loaded and un-

loaded at run-time by using the JAsCo run-time infrastructure API. In addition, JAsCo includes a very flexible system for remotely (from outside the application) adding a connector. At regular time intervals, JAsCo scans the classpath for new connectors. When a new connector has been found, it is automatically loaded in the system. As such, activating a connector in an application simply means placing the connector in the classpath of the application. Likewise, the removal of a connector is detected by the JAsCo run-time infrastructure and the connector and its instantiated aspect hooks are automatically removed from the system. Obviously, this system can be disabled if the performance penalty of scanning the classpath is considered too high. JAsCo also supports the instantiation of a hook on expressions that contain wildcards. These limited regular expressions are matched at run-time. Consequently, when a new component is added to an application, it is automatically affected by all aspects that were declared using wildcards. In addition, run-time wildcard matching makes the compilation of a connector that instantiates a wildcard aspect very fast, this in comparison to approaches that resolve these wildcards at compile-time. On the other hand, matching the wildcard expressions at run-time degrades the run-time performance. The main advantage of this trapped component model consists of the portability of the approach. JAsCo does not depend on a specialized virtual machine nor on some custom interfaces only available at certain systems. For example, a run-time environment optimized for embedded systems (JAsCoME) and an implementation of JAsCo for the .NET platform [35] have recently been proposed. The drawback of a trapped approach is of course that a performance overhead is experienced for all these traps, even if no aspects are applied.

4.2 JAsCo HotSwap

Applications equipped with traps using the JAsCo preprocessor 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 to all possible joinpoints. In [DAW 2004], we present the HotSwap framework to improve the performance of the interception part of the JAsCo technology. JAsCo 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. 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 API (JPLIS), which avoids running the virtual machine in debugging mode. The byte-code manipulations themselves are again implemented using the Javassist library. Implementing a HotSwap system for AOP is technically quite challenging. The main problem is that both JDI and JPLIS do not allow altering anything of a class besides the method bodies. For inserting traps, class schema changes are required because the original method is copied to a method with a different non-existing name. This method can then be invoked by the JAsCo run-time infrastructure when the original behavior is required (e.g. when `proceed` is invoked

in an around advice). Copying the original method also allows to easily and efficiently remove the trap as it suffices to rename the method copy to the original method name. In order to allow this without run-time schema changes, every class is still altered at load-time to add a dummy method for every method in the class. This dummy method is then able to contain the original method body when inserting a trap. In order to implement an efficient AOP system, several fields containing reflective data about the joinpoints contained in the class are also required. Because run-time schema changes are not allowed, these fields cannot be added to the class where the joinpoint belongs to. Therefore, a separate class containing all those fields is generated each time new traps are installed. As such, the JAsCo HotSwap implementation requires somewhat more memory at run-time than when traps are installed using the traditional preprocessor approach. Using the HotSwap framework, the JAsCo run-time infrastructure is able to install traps at only those methods that are subject to aspect application. When a new aspect is added, all the methods where the added aspect is applied upon, are hot-swapped at run-time with a trapped version. Because HotSwap does not allow replacing single methods, the complete class byte code is replaced with a version where the methods, upon which aspects are applied, are trapped. All other methods of the class however remain untouched. Likewise, the original method byte code is installed when the aspect is removed again and if no other aspect is applicable on the method at hand. Of course, 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 to be five to ten times slower than the statically compiled language AspectJ [DAW 2004, other benches, DAVY?]. This overhead is mainly caused by the additional indirection these traps impose. In addition, the traps are constant 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 huge performance gain can be realized by avoiding capturing the actual arguments.

4.3 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 joinpoints. 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 weaved joinpoint behavior might become invalid. This happens when a connector is added that instantiates a hook that is applicable on a joinpoint where aspects are already attached or when a connector is removed that contains an applicable hook for such a joinpoint. In addition, it is possible to change some properties of a connector dynamically so that the applicable context of the instantiated hooks is altered. The JAsCo run-time weaver is able to cope with these issues. When no advices are applicable any longer, the original byte code contained in the method copy is installed again.

Generating a weaved joinpoint is not always achievable because it is possible that whether a hook is applicable or not, has to be re-evaluated for every execution of a given joinpoint. For example, when a hook defines a cflow condition in its constructor, this constructor has to be re-evaluated for every execution of a joinpoint. However, the entire constructor does not have to be re-evaluated. In this case, only the result of the cflow condition is able to change for different executions of the joinpoint. As such, partial evaluation techniques can be used to cache a partially evaluated constructor. In addition, for the particular cflow construct, it is sometimes possible to statically analyze whether the condition might ever be true or not by examining the call graph of an application. This technique is elucidated in [27].

Another major optimization of the JAsCo run-time weaver consists of detecting which static and dynamic reflective joinpoint information the aspects might require. Suppose for instance that an aspect only requires the method name of the current joinpoint. 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 joinpoint 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 joinpoint only require the method name for instance, only the method name is provided. 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. However, in many cases, this analysis allows a major optimization.

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 joinpoint. 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 joinpoint 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 joinpoints that are executed more than twenty times in the past sec-

ond. JAsCo thus effectively combines and integrates three alternative aspect weavers.

4.4 Performance Evaluation

TODO: - short intro to approaches, best grouped: * AspectJ-AW compile-rt weaving *JBoss/AOP traps *Spring/AOP,dynAOP dyn proxies *cgLib??? - Explain AWBench - description of benches, JAsCo does better!!!! - notice logarithmic scale - performance converges (couple of nanoseconds difference for AW, JAsCo, AspectJ in most benchmarks are not significant), maybe limit of traditional weaving?

see figure 3, try to place figure here

5. IMPLEMENTING STATEFUL ASPECTS USING THE RUN-TIME WEAVER

As explained before, a naive approach for integrating the stateful aspect would be weaving it at all possible joinpoints defined within the protocol. This induces a performance overhead at all these joinpoints, while the stateful aspect is only interested in a limited set of joinpoints 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 joinpoints where the aspect is currently interested in. When a transition is fired, the weaver unweaves the aspect at the joinpoints associated with the current transition and weaves it back in at the joinpoints relevant for the subsequent transitions. As such, a real *jumping aspect* is realized. Notice that when the aspect *dies* because no subsequent transitions are defined, it is completely unweaved. 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 joinpoints 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 joinpoints and never unweaves it unless the aspect itself is manually removed or dies.

```
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)[8]. 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 joinpoint is encountered, the outgoing transitions of the current state are evaluated with the given joinpoint 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 rewaved to the new joinpoints 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.

6. RELATED WORK

TODO: organize, edit and maybe condense? Add Andrew or Babel? TODO: steamloom shorter, add PROSE,PROSE2, JAC, ...?

Apart from the approaches employed in our benchmarks, several other AOP approaches are introduced for enabling dynamic AOP. Event based aspect oriented programming (EAOP) allows specifying crosscutting concerns by employing event patterns which are described using a formal language [TODO]. Because of this formal model, advanced detection and resolution of aspect interactions becomes possible. On the implementation level, EAOP inserts traps that query a central execution monitor, similar to the JAsCo connector registry. The execution monitor has a global view of the executing application and contains all active EAOP artifacts. In contrast to JAsCo, EAOP inserts traps by source-code transformations.

Using Caesar [TODO], an aspect is described in terms of an Aspect Collaboration Interface (ACI). Each concrete aspect needs to implement the required methods specified by its corresponding ACI. Aspect bindings connect the aspect implementations to different concrete deployment contexts. One of the major contributions of the Caesar approach is the introduction of aspectual polymorphism. Aspect bindings are able to implement a binding for different types and the concrete binding is resolved dynamically using the type of the object at hand. In this viewpoint, aspectual polymorphism is similar to the concept of late binding found in object oriented languages.

Filman [TODO] 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.

Steamloom [TODO] is another dynamic AOP approach that supports the run-time weaving of aspects. Similar to PROSE2 [22], Steamloom aims at achieving an aspect-aware Java Virtual Machine in order to boost the run-time AOP performance, this in contrast to most other AOP technologies which implement their AOP facilities on top of the JVM. At the moment, Steamloom is implemented as an extension of IBMs Jikes Research Virtual Machine [12] and employs its adaptive optimisation 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, without requiring any pre-processing of the base application itself. The main advantage over a run-time weaver is that it avoids the weaving overhead, this makes it very suitable when aspects are applied and removed very frequently. Currently, the Steamloom technology supports the Caesar component model and language, although only before and after advice deployment is supported.

Finally, AspectS [TODO] introduces dynamic AOP support within the Squeak/Smalltalk environment. Pointcuts and their corresponding advices are described making use of plain Smalltalk. By sending the install and uninstall message to an instance of such an aspect, aspects are activated and deactivated within the application at run-time. At the implementation level, AspectS makes use of the dynamic properties of Smalltalk itself. In this case, Method wrappers are used which are placed around a compiled method by replacing its entry in the method dictionary of a class. This way, it is possible to easily add behavior, in this case aspect advices, to method invocations.

Wool [TODO] finally is a dynamic AOP approach that supports two types of dynamic weaving strategies. Similar to PROSE, the Wool system is able to employ the Java Debugging Interface to intercept the execution of the base application. However, aspects can also be inserted invasively into the target joinpoints by employing Java HotSwap. The original contribution of Wool is that aspects are able to implement their own heuristics for deciding whether they are invasively inserted or not. The 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.

Douence et al. propose a model for supporting stateful aspects [4] as an extension of their formal aspect model [3]. The advantage of this formal model is that it allows to automatically deduce possible malicious interactions between aspects. Furthermore, the model supports composition of stateful aspects using well-defined composition operators. A proof of concept implementation of this model is also available [5]. This implementation is however based on static program transformations and as such it requires to advice all possible joinpoints defined within the protocol. The JAsCo implementation improves on this because only a subset of the joinpoints are actually advised.

Walker et al. introduce *declarative event patterns* (DEPs) [13] as a means to specify protocols as patterns of multiple events. They augment AspectJ aspects with special DEP constructs (called *tracecuts*) that can be advised similarly to pointcuts. Their approach is based on context-free grammars, and involves a transformation of the DEP constructs into standard AspectJ aspects containing an event parser, similar to the transformation realized by parser generators in compiler technology. While DEPs can recognize properly nested events and thus possess an even higher degree of declarative expressibility than the JAsCo approach, they only provide for the ability to attach advice code to entire protocols. Separate transitions of the protocol cannot be advised, and several overlapping protocols (realized through several independent event parsers) would have to be employed to mimic this

possibility of JAsCo. 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. Also, there are some unresolved issues in the current implementation of DEPs regarding optimal conservation of relevant execution traces.

7. CONCLUSIONS

TODO: update conclusions with new paper.

In this paper we introduce an extension of the JAsCo language that enables triggering aspects on a sequence of joinpoints. The JAsCo stateful aspects extension allows to declaratively specify a regular protocol. Advices can be attached to each transition in the protocol. JAsCo also allows to trigger aspects on the complement of a protocol given a set of joinpoints. Because of the declarative specification, the stateful aspect is easier to understand and evolve than a manual implementation using singular joinpoints. In addition, the declarative specification allows to optimize the execution of the stateful aspect. By employing dynamic AOP, the stateful aspect behavior is only weaved at those joinpoints the aspect is currently interested in.

A limitation of the current approach is that JAsCo stateful aspects can only specify regular protocols. Protocols that require a non-regular language like for example $x \text{ times } A; B; x \text{ times } A$, cannot be represented. 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 joinpoints 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.

8. REFERENCES

- [1] J. Aycock and N. Horspool. Schrodinger's token. *Software Practice and Experience*, 31(8), 2001.
- [2] J. Brichau, W. De Meuter, and K. De Volder. Jumping Aspects. In *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.
- [3] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, Pittsburgh, USA, October 2002.
- [4] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Proceedings of the 3th International Conference on Aspect-Oriented Software Development*, Lancaster, UK, March 2004.
- [5] R. Douence, P. Fradet, and M. Südholt. Trace-based Aspects. *Aspect-Oriented Software Development*, September 2004.
- [6] T. Andrews et al. Business Process Execution Language for Web Services Specification, May 2003. <http://www.ibm.com/developerworks/library/ws-bpel/>.
- [7] A. Farias and M. Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In *Distributed Objects and Applications 2002*, Irvine, USA, October 2002.
- [8] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory*. Addison Wesley, 2st edition, 2001.
- [9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and G.W. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, Budapest, Hungary, June 2001.
- [10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
- [11] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, USA, March 2003.
- [12] A. S. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.
- [13] R.J. 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, November 2004.
- [14] D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

