Aspect-Oriented Programming using JAsCo

Wim Vanderperren, Davy Suvée, Bruno De Fraine, and Viviane Jonckers

System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels
Belgium
{wvdperre,dsuvee,bdefrain,vejoncke}@vub.ac.be

Abstract. In this paper, the JAsCo aspect-oriented programming language is introduced. JAsCo is tailored for component-based software development and introduces two new concepts: aspect beans and connectors. An aspect bean is highly reusable because it allows describing crosscutting concerns independently of concrete component types and APIs. A connector on the other hand, deploys one or more aspect beans within a concrete component context and allows specifying an explicit combination of their aspectual behavior. The JAsCo technology introduces a genuine run-time weaver that flexibly weaves, reweaves and unweaves aspects at run-time. By comparing the run-time performance of this weaver with other state-of-the-art dynamic AOP approaches, it is shown how JAsCo is able to significantly outperform most of these approaches.

1 Introduction

Component-Based Software Development (CBSD) and more recently Aspect-Oriented Software Development (AOSD) have been proposed to tackle problems experienced during the software engineering process. When applying CBSD, a full-fledged software-system is developed by assembling a set of pre-manufactured components. Each component is a black-box entity that can be deployed independently and that delivers one or more specific services to the system [1, 2]. The deployment of this paradigm drastically improves the speed of development and the quality of the produced software. AOSD on the other hand, aims at improving the separation of concerns in current software engineering methodologies, by providing an extra separation dimension for decomposing a software system [3, 4].

Originally, aspect-oriented research and practice mainly focused on the object-oriented paradigm, where it claims that the class hierarchy is insufficient for the clean modularization of all possible concerns (a problem dubbed "the tyranny of the dominant decomposition" in [4]). Component-based software development however also suffers greatly from crosscutting concerns and tangled code, since it advocates low coupling between components and high cohesion of single components in order to make components reusable and independently deployable [2].

As a result, a lot of functionalities are spread and repeated over different components making them difficult to comprehend and maintain. Hence, integrating the aspect-oriented ideas into the component-based world, significantly contributes to the component-based paradigm as it allows increasing the modularity of component-based applications.

The other way around, namely the adoption of component-based ideas within AOSD, is a valuable concept as well, and we claim that current mainstream aspect-oriented approaches such as AspectJ [5] can tremendously benefit from this integration. In previous work [6], we introduced the JAsCo approach, an aspect-oriented language tailored for component-based software development in a Java setting. JAsCo is inspired by the integrated Aspectual Components approach [7], and features reusable and independent aspect entities that are deployed upon a base application employing a separate composition process. However, JAsCo also combines this design with the straightforwardness of the AspectJ model, where aspects describe pointcuts and advice code that must be applied at their resolved joinpoints. We believe that the JAsCo approach is therefore particularly well-suited to illustrate how the adoption of component-based ideas contributes back to the AOSD field, and we present its design and implementation from this perspective in this paper.

The concrete improvements that JAsCo brings about by applying component-based concepts are diverse. *Aspect beans*, JAsCo's aspect entities, are modeled after Java Bean components and are deployed upon a concrete base application through separate *connectors*. This design enables a higher degree of reuse and consequently entails faster development. Furthermore, by equipping connectors with flexible *precedence and combinations strategies*, JAsCo also allows specifying how deployed aspects should be combined in case of feature interaction [8], i.e. when multiple aspects advise the same joinpoint. Component-based software development emphasizes the need for run-time reconfiguration, which means that individual components as well as complete component compositions can be altered or replaced at run-time. This is in sheer contrast to AOSD's original focus on statically deployed aspects with limited instantiation and static joinpoints (i.e. aspects that advise points in the compile-time structure of the program). More dynamism for aspects has been long argued for however [9–11], and the dynamic joinpoint model, where advices are triggered by run-time events, has become commonplace. JAsCo further improves on this by providing flexible *aspect factories*, *run-time deployment* of aspects and so-called *stateful pointcuts* that match complete sequences of events rather than singular events. Parallel innovations in the JAsCo implementation, such as run-time (re)weaving, make these dynamic features possible while retaining performance characteristics comparable to static approaches.

This paper is structured as follows. The following section introduces the main JAsCo concepts and presents the language elements. It also compares these concepts with the current state of the well-known AspectJ approach. Section 3 presents the JAsCo technology and discusses several approaches to improve this technology performance-wise. In section 4, the performance of this enhanced

JAsCo technology is compared with other state-of-the-art AOP approaches. Afterwards, research that is related to this work is summarized. Finally, we state our conclusions and describe future research directions.

2 The JAsCo Model

In this section, we present the concepts and features introduced by the JAsCo aspect-oriented programming model. Similar to AspectJ, JAsCo proposes an asymmetric AOP model, where crosscutting concerns are modularized as dedicated entities, called *Aspect Beans*. The JAsCo aspect bean implementation does however not refer to concrete component types and APIs, as such making aspect beans reusable amongst multiple applications. In order to deploy an aspect bean, we employ a *Connector* construct, as proposed by Aspectual Components, which instantiates a reusable JAsCo aspect bean definition within a concrete component context. In order to illustrate the JAsCo AOP model, we employ a small case study that introduces a basic security concern that ensures that a user has the required credentials for accessing particular functionalities offered by a software system. Access control is a typical example of a crosscutting concern [12] as its implementation logic is tangled and spread amongst various components of the system that requires authentication. The next two subsections introduce the concepts of the JAsCo aspect/connector model and its language in more detail. Afterwards, we illustrate the JAsCo solution for combining the behavior of separately specified aspects which are applicable at the same points within the application. Finally, we propose an extension of the JAsCo model that supports the specification of stateful aspects, which describe pointcuts in terms of a sequence of events instead of single events.

2.1 JAsCo Aspect Beans

In order to modularize a crosscutting concern, JAsCo introduces the *Aspect Bean* construct. In essence, an aspect bean can be considered as an extension of a regular Java Bean. As such, its syntax and concepts remain close to the ones introduced by regular Java. A JAsCo aspect bean contains two parts:

- Any number of **ordinary Java class members**, such as data fields, methods and inner classes.
- One or more **hook definitions**, which are responsible for capturing the crosscutting behavior.

Listing 1 illustrates the aspect bean modularization of the crosscutting access control concern. The aim of this concern is to validate whether a user possesses the required credentials to access particular functionalities offered by a software system. The `AccessManager` aspect bean declares two ordinary Java data fields: one provides access to the permissions database (line 3) and one contains the current system user (line 4). All data fields and methods declared by an aspect

bean are shared amongst the contained hooks. For implementing the crosscutting behavior, an aspect bean groups together one or more logically related *hook definitions*. A hook[1] can be considered as a special kind of inner class and contains the following three parts:

- A hook **triggering condition** specification, build up out of:
  - A *constructor*, specifying a declarative abstract pointcut definition.
  - An `isApplicable` method, specifying an additional triggering condition employing full Java.
- One or more **advice methods**.
- Any number of **ordinary Java class members**, which are local to the hook.

Instead of hard-wiring the concrete deployment context within the aspect bean implementation itself, we propose a special constructor that describes a pointcut in a more abstract way. Each constructor takes several *abstract method parameters* as input, which serve as placeholders for concrete method signatures that are provided when a hook is instantiated within a specific component context. These abstract method parameters are employed within the constructor body that outlines how the joinpoints of an instantiated hook should be computed. The purpose of the `AccessControl` hook (lines 6-29) is to throw an exception whenever the current system user accesses some functionality without possessing the required permissions. The constructor of this hook (lines 10-12) declares one abstract method parameter, `method`, that takes zero or more arguments as input. The scope of these method arguments extends the entire hook. Hence, arguments can be immediately employed within the hook advice methods, while other AOP approaches requires to explicitly capture them. The constructor body (line 11) specifies that the behavior of the hook should be triggered when the method bound to the abstract method parameter `method` is *executed*. Hence, when a hook is instantiated using concrete method signatures later on, the joinpoints of the hook are resolved to the execution of these concrete methods. Similar to AspectJ, JAsCo is able to specify more elaborated pointcut definitions in the constructor body, for instance control flow conditions that are combined using logical operators. In some cases, the triggering of the hook behavior requires additional logic which can not be expressed in standard declarative pointcut conditions. The authentication behavior of the `AccessControl` hook for instance, should only be triggered if the current user is not the *root* user. To this end, JAsCo provides the `isApplicable` method, similar to the `if` construct in AspectJ, that allows to describe an additional triggering condition by employing the full expressiveness of Java. The `isApplicable` method of the `AccessControl` hook (lines 14-16) evaluates to true if the current user is not the root user.

[1] The term “hook” is overloaded, as in an AOP-context, “hooks” are generally considered to be shadow joinpoint decorators. The term “hook” was however adopted early on, making it difficult to change it now.

```
1  class AccessManager {
2
3      PermissionDb pdb = new PermissionDb();
4      User currentuser = null;
5
6      hook AccessControl {
7
8          String exceptionmessage = "General Access Exception";
9
10         AccessControl(method(..args)) {
11             execution(method);
12         }
13
14         isApplicable() {
15             return !pdb.isRoot(currentuser);
16         }
17
18         around() {
19             if(pdb.check(currentuser,thisJoinPointObject))
20                 return proceed();
21             else
22                 throw new AccessException(exceptionmessage);
23         }
24
25         void setExceptionMessage(String aMessage) {
26             exceptionmessage = aMessage;
27         }
28
29     }
30 }
```

Listing 1. The JAsCo access control aspect bean

Hook advice methods are employed for implementing the various actions a hook needs to perform whenever one of its computed joinpoints is encountered. JAsCo supports several types of advices, including `before`, `around`, `after`, `after throwing` and `after returning`, from which the semantics match their AspectJ counterparts. Hook advice methods are implemented using the full Java expressiveness and only one implementation for each advice type can be provided. The `AccessControl` hook defines an `around` advice that implements the access control behavior. An around advice wraps the original execution of a joinpoint and allows to explicitly specify whether the behavior of the encountered joinpoint should still be executed. The `around` advice method of the `AccessControl` hook (lines 18-23) makes sure a user possesses the required credentials, by checking whether the object associated with the current joinpoint (referenced by `thisJoinPointObject`) can be accessed by the user of the system. If this is not the case, an exception is thrown, from which the messages can be customized by employing the `setExceptionMessage` method (lines 25-27). In the other case, the original joinpoint execution is continued by explicitly invoking the `proceed` method. When multiple hooks are applicable at the same joinpoint, the `proceed` method continues by invoking the around advice of the the next hook, this way building up a chain of around advices.

The `AccessManager` aspect bean does not hard-wire specific deployment information (i.e. component types and APIs) within its implementation. Hence, this concern can be reused within various applications that require authentication. Next to making an aspect reusable, the abstract method parameters of

```
1  class AccessManagerErrorPolicy extends AccessManager {
2
3      hook AccessControlErrorPolicy extends AccessControl {
4
5          around() {
6              if(pdb.check(currentuser,thisJoinPointObject)) {
7                  return proceed();
8              } else {
9                  performErrorPolicy(currentuser,thisJoinPointObject);
10             }
11         }
12
13         refinable void performErrorPolicy(User user, Object object);
14
15     }
16 }
```

Listing 2. The JAsCo access control aspect bean with refinable error policy method

a hook constructor also make it possible for JAsCo to guarantee type safety for arguments and return values of abstract method parameters. Consider a hook constructor with an abstract method parameter that expects an object of type `Integer` as first argument. When this hook is instantiated with a concrete method signature, this method is type checked against the abstract method parameter by employing the regular Java typing rules. This allows to identify possible type incompatibilities that would otherwise lead to run-time errors. This kind of type safety is not provided by typical framework-based AOP approaches, such as JBoss/AOP [13] and Spring/AOP [14] where connections are specified in XML descriptors and advices need to include possible type unsafe casts in order to implement context specific behavior.

By default, the `AccessControl` hook throws an exception when a user accesses some component functionality without possessing the required permissions. Sometimes however, a deployer of the access control aspect could envision a custom access error policy, for instance mailing the system administrator or logging the access error to file. In order to preserve the reusable nature of the access control aspect, but at the same time support custom access error policies, *refinable* methods can be employed. A refinable method, which is similar to an abstract method in Java, allows to postpone the implementation of certain aspect behavior in order to remain context independent. Listing 2 illustrates an extended version of the `AccessManager` aspect bean that supports custom error policy handling. JAsCo inheritance is employed to extend and override the behavior of the `AccessManager` aspect bean and its corresponding hooks: the `AccessManagerErrorPolicy` hook inherits the behavior of the `AccessControl` hook (line 3), but overrides the original around advice (lines 5-11) so that it invokes the refinable `performErrorPolicy` method (line 13) for handling access errors. A context-specific implementation should be provided for the `performErrorPolicy` method when the `AccessManagerErrorPolicy` hook is deployed within a concrete component context, this by in-lining the method implementation within the JAsCo connector.

```
1 connector PrintAccessControl {
2
3    AccessManager.AccessControl control =
4       new AccessManager.AccessControl(void Printer.printFile(File));
5
6    control.setExceptionMessage("Printer Access Exception");
7    control.around();
8 }
```

Listing 3. The JAsCo connector for print access control

2.2 JAsCo Connectors

In order to deploy abstract and reusable aspects beans within a concrete component context, JAsCo introduces the *connector* construct. The purpose of a connector is threefold:

- **Instantiate** one or more logically related aspect bean hooks (defined amongst multiple aspect beans).
- **Initialize** aspect bean hook instantiations with context-specific properties.
- Define **precedence** for advice method executions and deploy **combination strategies** to resolve conflicting aspect interactions.

Similar to instantiating regular Java classes, aspect bean hooks are instantiated by making use of their corresponding constructors. Reconsider the previously introduced case-study: a `Printer` component should only be accessed by users who have the appropriate printing permissions. Listing 3 illustrates a connector that instantiates the `AccessControl` hook of the `AccessManager` aspect bean with the `printFile` method signature of the `Printer` component (lines 3-4). As a result, the `printFile` method of the `Printer` component gets bound to the `method` abstract method parameter of the `AccessControl` hook. Hence, the abstract pointcut described by the `AccessControl` hook constructor body is resolved to one specific joinpoint, namely the execution of this `printFile` method. The exception message thrown in case of access violations is customized by invoking the `setExceptionMessage` method on the `control` hook instance (line 6). Additionally, a connector can specify the advice methods to execute whenever a resolved joinpoint is encountered. The `PrintAccessControl` connector declares the execution of the `around` advice method (line 7). By default, if no advice method executions are specified for a particular hook instance, all advice methods implemented within the hook are executed. The deployment of the `PrintAccessControl` connector has following implication upon the system: whenever the `printFile` method of the `Printer` component is executed by a user who does not possess root permissions, his/her credentials are verified and an access exception is thrown accordingly.

Listing 4 illustrates the instantiation of the `AccessControlErrorPolicy` hook, which supports custom error handling implemented as a refinable method. Hooks that declare refinable methods are instantiated like regular hooks (lines 3-5), but a concrete implementation for their refinable methods is in-lined within their instantiation. The in-lined implementation for the `performErrorPolicy` method, informs the administrator about access violations of its system users

```
1  connector PrintFaxAccessControlWithErrorPolicy {
2
3     AccessManagerErrorPolicy.AccessControlErrorPolicy control =
4        new AccessManagerErrorPolicy.AccessControlErrorPolicy (
5           { * Printer.*(*) , * Fax.*(*) } )
6           {
7              void performErrorPolicy(User user, Object object) {
8                 Mailer.mailAdmin("Error: " + user + " - " + object);
9              }
10          }
11
12       control.around();
13 }
```

Listing 4. The JAsCo connector for print access control with custom error handling policy

by mail (lines 7-9). When instantiating a hook, it is possible to bind multiple method signatures with the same abstract method parameter by inserting them between braces. Additionally, wild-cards can be employed to easily select all methods defined within multiple components without having to enumerate them. Hence, the same instance of the `AccessControlErrorPolicy` hook is employed to enforce access control upon all functionalities offered by the `Printer` and `Fax` components.

JAsCo Aspect Factories Although the pointcut expression described by a hook constructor can be resolved to multiple joinpoints, only one instance of a hook is instantiated by its constructor. If one requires a hook instance for each resolved joinpoint, separate hook instantiations are required. In order to support the automatic generation of multiple hook instances within a single instantiation expression, JAsCo provides a set of aspect factory constructs, such as `perall`, `permethod`, `perclass`, `perinstance` and `perthread`, which are declared in front of a hook instantiation. The `perall` factory generates a unique hook instance for each resolved joinpoint execution. The `permethod` factory generates a unique hook instance for each method that is resolved as joinpoint shadow. In the context of our case-study, the `perclass` factory could be employed to generate a separate access control hook instance for each resolved component type that requires authentication. Several aspect factories can be combined to obtain more expressive hook instantiations. Combining aspect factories generates the union of the instances generated by the factories separately. For instance, combining `perinstance` and `perthread` generates a unique hook instance for every object instance and executing thread. If the expressive power of the predefined factories is not sufficient, one can define custom aspect factories by implementing the `IAspectFactory` interface. These custom aspect factories are modular plugins for the JAsCo weaver and are defined in front of a hook instantiation using the `per` keyword.

2.3 Aspect Combinations

JAsCo aspect beans do not only allow to describe crosscutting concerns independent from a specific deployment context, but also independent from other

```
connector PrintLockAccessControl {

   AccessManager.AccessControl acontrol =
      new AccessManager.AccessControl(* Printer.*(*));

   LockingManager.LockControl lcontrol =
      new LockingManager.LockControl(@ThreadUnsafe * Printer.*(*));

   lcontrol.around();
   acontrol.around();
}
```

Listing 5. Connector that controls the precedence of hooks. Also notice the use of the novel Java 1.5 meta-data feature for limiting the set of joinpoints for the `LockControl` hook instance.

aspects that are deployed within the same application. In some cases however, it should still remain possible to describe how several aspects should cooperate when they are applicable at the same joinpoint. This kind of *feature interaction* is common to several engineering disciplines, such as telecommunication systems [15], and has already been identified as a recurring problem in AOP [8]. To allow the specification of a resolution for possible aspect interactions, JAsCo introduces an expressive aspect combination mechanism based on *precedence* and *combination strategies*.

JAsCo Precedence Strategies Using JAsCo, it is possible to instantiate multiple hooks within the same connector. When no advice method executions are specified within this connector, the advice methods are triggered in the order in which their hooks were instantiated. JAsCo however also allows defining an explicit execution sequence by employing precedence strategies. Listing 5 illustrates the simultaneous deployment of a *lock* and *access control* aspect upon all methods defined within the `Printer` component. The lock aspect implements a primitive synchronization concern that is able to set an exclusive lock onto the some functionality offered by a component. Whenever mutual joinpoints of the `acontrol` and `lcontrol` hook instances are encountered, the precedence strategy, specified using the advice method executions, is applied. In this case, the access to the `Printer` component is to be locked first: this is specified by the execution of the *around* advice method of the `lcontrol` hook (line 9). When an exclusive lock is obtained, the *around* advice method of the `acontrol` hook instance is executed to verify whether the system user possesses the required permissions to use the `Printer` component (line 10). Note that JAsCo precedence strategies are instance based, i.e. it remains possible to specify a different precedence strategy for other instantiations of the same hooks.

Similar to defining precedence for advice method executions within a connector, JAsCo also provides a solution for defining precedence amongst connectors. In general, connectors should instantiate one or more hooks which together compose a logically related unit. If one however wants to define precedence amongst hooks which are not logically related, he/she is forced to instantiate these hooks in a single connector. In order to preserve clean modularization, JAsCo allows to attribute priorities to connectors, which assure that the behavior of hooks,

```
interface ICombinationStrategy {

    public HookList validateCombinations(HookList);

}
```

Listing 6. The interface for combination strategies

contained within a connector attributed with a higher priority, are executed first. Furthermore, these priorities can be dynamically adapted in order to adhere to the changing requirements of an application.

JAsCo Combination Strategies Precedence strategies provide a solution for a limited set of feature interaction problems, as some complex aspect combinations require a more expressive way of declaring how their behavior should cooperate. One might for instance be interested in excluding the behavior of aspect B, if the behavior of aspect A is also applicable at the same joinpoint. This interaction could be described by introducing an additional connector keyword *excludes*, which specifies that aspect A excludes the behavior of aspect B. Other aspect combinations however, require additional keywords and it is impossible to envision all possible aspect combinations in advance. Therefore, JAsCo proposes a more flexible and extensible system that allows defining custom combination strategies implemented using the full Java expressiveness. Each concrete combination strategy implements the `ICombinationStrategy` interface shown in Listing 6. In general, a JAsCo combination strategy can be considered as a filter that acts on the list of applicable hooks (hooks where both the constructor and the `isApplicable` method evaluate to true).

Consider a change in requirements within our case-study: all accesses to component functionalities need to be logged for future reference. This logging should however not be triggered for actions performed by the root user. Hence, the logging aspect should only be triggered if the access control aspect is active at the same joinpoint. The `TwinCombinationStrategy` of Listing 7 illustrates the implementation of this aspect combination. Each concrete combination strategy implements the `validateCombinations` method that filters the list of applicable hooks and possibly modifies their internal behavior. The constructor of the `TwinCombinationStrategy` instantiates this strategy with two hook instances (lines 5-8). The `validateCombinations` method implements the removal of `hookB` if `hookA` is not encountered in the list of applicable hooks (lines 10-15). As a result, the advice methods of `hookB` are never executed if `hookA` is not found in the list of applicable hooks.

Listing 8 illustrates the deployment of the `TwinCombinationStrategy`. The `LoggingAccessControl` connector instantiates both the `AccessControl` and `FileLogger` hooks with all methods defined by the `Printer` component (lines 3-7). Both hook instances are used as input for the `TwinCombinationStrategy` instance (lines 9-10), which is added to the list of combination strategies by employing the `addCombinationStrategy` method. Hence, the logging behavior of the `FileLogger` hook is only applied if the behavior of the `AccessControl` hook

```
class TwinCombinationStrategy implements ICombinationStrategy {

    private Object hookA, hookB;

    TwinCombinationStrategy(Object a,Object b) {
        hookA = a;
        hookB = b;
    }

    HookList verifyCombinations(HookList hlist) {
        if (!hlist.contains(hookA)) {
            hlist.remove(hookB);
        }
        return hlist;
    }

}
```

Listing 7. The twin combination strategy

```
connector LoggingAccessControl {

    AccessManager.AccessControl control =
        new AccessManager.AccessControl(* Printer.*(*));

    Logging.FileLogger logger =
        new Logging.FileLogger(* Printer.*(*));

    addCombinationStrategy(new
        TwinCombinationStrategy(control,logger));

}
```

Listing 8. Connector deploying the twin combination strategy

is also applicable. Multiple combination strategies can be added to the same connector: the filtered list of applicable hooks of the first combination strategy is then passed on as input to the second combination strategy and so on.

2.4 JAsCo Stateful Aspects

Most aspect-oriented approaches offer facilities to express aspects in terms of dynamic joinpoints, such as run-time types, call-stack contexts (e.g. the `cflow()` pointcut) and dynamically evaluated expressions. Describing the applicability of aspects in terms of a *sequence* or *protocol* of joinpoints is generally not supported. With the exception of the `cflow` pointcut, pointcuts cannot refer to the history of previously matched pointcuts in their implementation. Douence et al. [16–18] propose a formal model for aspects with general protocol-based triggering conditions, called *stateful aspects*. In this section, we illustrate how JAsCo supports the specification of stateful pointcuts.

Reconsider the previously introduced case-study. At the moment, the access control aspect monitors all system activity, even if no specific user is logged in into the system. This however induces a certain run-time overhead which should be avoided. The implementation of the access control aspect, introduced in Listing 1, needs to be extended in such a way that its behavior is only triggered when a user is actually logged in. Hence, the access control aspect should start checking access permissions when a user logs in and stop checking access permissions

```
1  class StatefulAccessManager extends AccessManager {
2
3     hook StatefulAccessControl {
4
5        StatefulAccessControl(starttrigger(..a1),method(..a2),stoptrigger(..a3))) {
6           start>p1;
7           p1: execution(starttrigger) > p3||p2;
8           p2: execution(method) > p3||p2;
9           p3: execution(stoptrigger) > p1;
10       }
11
12       isApplicable p2() {
13          return !pdb.isRoot(currentUser);
14       }
15
16       around p2() {
17          if(pdb.check(currentuser,thisJoinPointObject)) {
18             return proceed(); }
19          else { throw new AccessException(); }
20       }
21    }
22 }
```

Listing 9. The stateful access control aspect bean

when the user logs out again. To implement this particular behavior, a stateful triggering condition is required. Listing 9 illustrates the improved implementation of the access control aspect bean which takes a start and stop condition into account.

The implementation of the `StatefulAccessControl` hook has many similarities with the original implementation of the access control aspect. The constructor (lines 5-10) takes three abstract method parameters as input (`starttrigger`, `method` and `stoptrigger`). The constructor body describes a protocol-based pointcut expression by enumerating a set of transitions from which the first one is highlighted with the `start` keyword. Each transition is labeled with a name (e.g. `p1`), defines a JAsCo compatible pointcut (e.g. `execution(starttrigger)`) and specifies one or more destination transitions that are matched after the current transition is fired. A transition fires when its corresponding pointcut expression evaluates to true. For example, the `p1` transition is only fired when one of the concrete method(s) bound to the abstract method parameter `starttrigger` are executed. In that case, transitions `p3` and `p2` are activated and evaluated for subsequent joinpoints encountered during the application execution. Note that the destination transitions are evaluated in the sequence defined in the destination expression. As such, when both the `p3` and `p2` transitions are applicable for a given joinpoint, only the `p3` transition is fired and only the `p3` destination transitions are evaluated for subsequent encountered joinpoints. This allows to keep the protocol deterministic and efficient to execute. In total, the constructor of the `StatefulAccessControl` hook describes the following stateful pointcut: the protocol commences in transition `p1`, where it waits for one of the concrete methods bound to the abstract method parameter `starttrigger` to be executed. When this event takes place, the stateful pointcut moves on to transition `p2` (line 7). The stateful pointcut remains in transition `p2` (line 8) while the concrete methods bound to the `method` abstract method parameter are being

```
1  connector SessionPrintAccessControl {
2
3     perthread StatefulAccessManager.StatefulAccessControl control =
4        new StatefulAccessManager.StatefulAccessControl(
5           void System.Login(User), * *.*(*), void System.Logout(User)
6              );
7
8  }
```

Listing 10. The JAsCo connector for stateful access control

executed and only moves on to transition `p3` if a concrete method bound to the `stoptrigger` abstract method parameter is executed (line 9). At that moment, the stateful pointcut returns to transition `p1`.

Similar to regular aspect beans hooks, stateful hooks are able to describe `isApplicable` methods that again specify an additional triggering condition. This method can either be applied upon all transitions or can be delimited to specific ones. In this case, an `isApplicable` method is specified for transition `p2`, as the condition whether a user is root should only be verified at the moment a user is actually logged in. Likewise, the scope of advices can be valid for all transitions or limited to specific ones. The around advice is applicable for transition `p2` and its behavior is only triggered if transition `p2` is triggered.

Listing 10 illustrates the deployment of the `StatefulAccessControl` hook. The `Login` and `Logout` methods of the `System` component are bound to the `starttrigger` and `stoptrigger` abstract method parameters of the access control hook. Hence, access control starts at the moment a user logs in and stops at the moment a user logs out again. At that moment, the hook is idle and waits for the execution of the `Login` method to restart its access control behavior. Also notice the `perthread` factory instantiation to make sure that every executing thread is associated with a unique access control hook. Otherwise, when multiple users are present in the system, the access control aspect fails[2].

JAsCo hook constructors allow to cleanly encapsulate stateful pointcuts. When state-based triggering conditions are not natively supported, state information is maintained within the advices, polluting the main aspect logic. In addition, explicitly specifying stateful triggering conditions allows to optimize the aspects performance-wise, as the run-time platform has knowledge about which joinpoints the aspect is currently interested in.

2.5 Comparison with AspectJ

In this section, we validate the claimed improvements of the JAsCo approach through a comparison with the AspectJ model in its current state. As one of the first and best-known aspect-oriented approaches, AspectJ is a suitable reference point for both the academic and industrial poles of the AOSD-field. We carry out this comparison by developing a small tracing example in AspectJ, while focusing on the aforementioned features such as aspect reuse, dynamism and the resolution of aspect interactions. We show that JAsCo offers certain benefits in these areas that cannot be achieved by AspectJ's model.

[2] Assuming that every new session or at least every user runs its own execution thread.

```java
public abstract aspect AbstractTrace {
    abstract pointcut TracePoint(Object obj);

    abstract void record(String action, Object obj);

    before(Object obj): TracePoint(obj) {
        record("Entering",obj);
    }

    after(Object obj): TracePoint(obj) {
        record("Leaving",obj);
    }
}
```

Listing 11. AspectJ abstract tracing aspect

```java
import java.io.PrintStream;

public abstract aspect PrintStreamTrace extends AbstractTrace {
    protected PrintStream output;

    public void setOutput(PrintStream p) {
        this.output = p;
    }

    public void record(String action, Object obj) {
        output.println(action + " on " + obj.toString());
    }

    abstract pointcut FlushPoint();

    after(): FlushPoint() {
        output.flush();
    }
}
```

Listing 12. AspectJ implementation of tracing on PrintStream output

AspectJ offers the concepts of abstract pointcuts and aspect inheritance as its means for aspect reuse. Listing 11 illustrates an abstract aspect that captures very general tracing behavior: at certain points exposing an object, we record the entering and leaving of that point. Several implementations of the recording behavior are possible. In Listing 12, we extend the abstract aspect with an implementation that records the tracing events on a `PrintStream` output. In this case, the inheritance relationship is very similar to inheritance between JAsCo aspect beans, as both are clearly modeled after standard Java inheritance for classes: new members can be introduced, and abstract members can be implemented or existing members can be overridden. Since we want to be able to use the `PrintStream` implementation of Listing 12 in several contexts, we do not bind the `TracePoint` pointcut in terms of a concrete API yet, and AspectJ therefore requires the aspect to remain abstract. Also note that besides adding new methods and fields, the specialization can include additional aspectual behavior, as is the case with aspect bean specialization in JAsCo. In our example, we might want to control the flushing of the output explicitly. To this end, the aspect in Listing 12 defines a new pointcut `FlushPoint`, together with an advice that specifies to flush the output after joinpoints are encountered that match

```
1 public aspect TraceDeploy extends PrintStreamTrace {
2    pointcut TracePoint(Object o): execution(* MyClass.*(..)) && this(o);
3
4    pointcut FlushPoint(): execution(* MyClass.majorMethod(..));
5
6    TraceDeploy() {
7       setOutput(Application.logStream);
8    }
9 }
```

Listing 13. AspectJ deployment of tracing behavior

this pointcut. Again, the pointcut is kept abstract to allow the deployment of the aspect in different contexts.

Whereas the usage of aspect inheritance appeared natural in the previous case because of the strong similarity with standard Java inheritance, this is no longer the case in Listing 13, where we deploy the `PrintStreamAspect` in a concrete application context. Firstly, this deployment takes place through the same concept (i.e. inheritance) in AspectJ, leaving no immediate or explicit distinction between deployment and specialization. As a matter of fact, the separation between these two phases is entirely the programmer's responsibility. In many cases this distinction is not made and aspects are implemented straight away with concrete pointcuts in mind that refer to a concrete deployment context. JAsCo is entirely different in this respect: by employing a separate connector construct for deployment, it automatically enforces all aspects to be independent of a concrete context, as they cannot define concrete pointcuts. Secondly, it seems at least awkward in the case of the `TraceDeploy` aspect to use the concept of inheritance for the deployment of an aspect, as opposed to e.g. instantiation. In AspectJ, aspects are not directly instantiated, but aspect instances are automatically created. As a result, an important variation point is lost in comparison to OO programming, namely the ability to customize an instance and thus vary among instances of the instantiated class. Inheritance is used to compensate for this loss in variation, or, put differently, the AspectJ model discourages having different instances of the same aspect, in favor of having different subaspects with one instance each. To set the output stream of our application in Listing 13, we are consequently forced to use the parameterless constructor of a subaspect[3]. We argue that JAsCo offers a model that is closer to existing OO concepts: as the hook instance is available for customization in the connector, it is possible to variate among aspects by calling the `setOutput` method on this instance, without the need for the introduction of a subtype. More generally, the entire deployment is performed through instantiation and reference in a connector rather than through specialization of aspect beans. This is because JAsCo models the

[3] Alternatively, for configuring aspect instances, it is possible in AspectJ not to create a subtype and to set the output from outside of the aspect instead by fetching the aspect instance through the static `aspectOf` method. It is however unclear where this code should be placed in the absence of a separate deployment construct. Placing it in the base code of the application is undesirable as AOSD advocates that the base application should be oblivious to aspects [19].

binding of the abstract pointcut to concrete context APIs as an instantiation of a hook.

We note another structural problem with the AspectJ model when we consider the resolution of feature interactions. The only resolution mechanism provided by AspectJ is the ability to influence the execution order of advices that apply at the same join point. This order is derived from a precedence concept between advices, which can in turn be influenced by the relative place in the aspect hierarchy or in the aspect definition, or by an explicit declaration of the precedence order between aspects. JAsCo provides several improvements over this setup. The resolution includes the flexibility of combination strategies in addition to ordering, and the ordering mechanism of AspectJ has been criticized [20] as complex and unintuitive, whereas JAsCo allows a straightforward specification in one place (the connector). Furthermore, the specification of the resolution is entirely part of the deployment, and can vary among different deployments, which improves the reuse possibilities of the aspect beans. This is not possible in AspectJ, as a precedence declaration couples the aspects and imposes the same resolution order for all deployments of the aspect. We observe another benefit of having a separate deployment construct: in AspectJ, since an abstract aspect must be extended in order to deploy it, a subaspect can only deploy one abstract aspect, at least in the absence of multiple inheritance. By allowing multiple logically related aspects to be deployed at the same time in a separate entity, JAsCo provides a well-suited place for the specification of resolution strategies between these aspects.

3 JAsCo technology

As stated in the introduction, the technology that realizes JAsCo has to be able to flexibly add and remove aspects at run-time. Furthermore, this technology has to be as portable as possible to support a wide range of possible application domains. To make this possible, we propose to equip every possible joinpoint with a trap[4] that enables the execution of aspect behavior. This way, attaching and removing aspects to trapped components does not require any adaptation whatsoever to the target components. A preprocessor tool is available that inserts these traps using the byte-code adaptation library Javassist [21]. Javassist supports both a very high level API, that allows inserting plain Java code, and a low level API, that manipulates byte-code directly. As such, using the high level API, rapid prototyping is possible but when necessary, performance critical parts can be replaced with low-level byte code manipulations.

Each trap refers to the JAsCo run-time infrastructure that manages the registered connectors and aspect beans. Fig. 1 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

[4] The term trap is often called *hook* in other AOP approaches. We use trap to avoid confusion with the hook language construct.

run-time, the connector registry is notified and its database of registered connectors and hooks is updated dynamically. The left-hand side of Fig. 1 illustrates a JAsCo-enabled component from which the methods are equipped with traps. As a result, whenever a method is called, its execution is deferred to the connector registry. The main method of communication of Java Beans is event posting, so firing an event also reschedules execution to the connector registry. When a trap is encountered, the connector registry looks up all connectors that are 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.

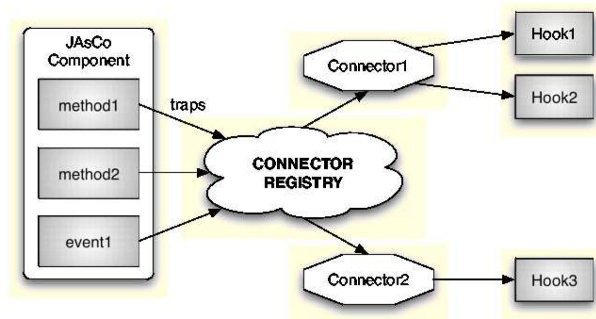

Fig. 1. JAsCo run-time architecture

Our approach is very flexible to support unanticipated run-time changes. When a new component is added at run-time, it is automatically affected by all aspects that are instantiated using the appropriate pointcut description. Connectors can also 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[5] 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 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.

The main advantage of this trapped model consists of the portability of the approach. JAsCo does not depend on a specialized virtual machine nor on 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 [22] have been proposed. Of course, the drawback of this approach is the experienced performance overhead for all traps, even if no aspects are applicable.

[5] It is also possible to specify a separate connector loadpath where JAsCo has to search for connectors.

```
1 public void executeJoinpoint(Joinpoint jp) {
2     hook1.before(jp);
3     hook2.before(jp);
4     hook0.around(jp);
5     hook1.after(jp);
6 }
```

Listing 14. Simplified Java counterpart of the cached combined aspectual behavior at a joinpoint. The `hook0-2` variables are initialized with the correct hooks when creating the class

3.1 Jutta

It is no surprise that the dynamic features offered by JAsCo induce a substantial run-time overhead. The overhead induced by JAsCo in real-life applications is unacceptably high in comparison to hard-coding the advices in the base program. This high overhead is mainly caused by the JAsCo run-time infrastructure which acts as an aspect interpreter. For each joinpoint, JAsCo evaluates which hooks are applicable. When no connectors are added or removed, the set of applicable hooks remains unchanged for every joinpoint. As such, when the same joinpoint is encountered several times, the algorithm for finding the appropriate hooks and executing their behavior is executed over and over again. Therefore, a huge performance gain can be achieved by compiling and caching the combined aspectual behavior for often encountered joinpoints. Although this compilation process requires some overhead, this optimization pays off when when a joinpoint is frequently encountered. In fact, this strategy is similar to just-in-time compilers [23] used in modern virtual machines and therefore our approach is named *Jutta* (Just-in-time combined aspect compilation).

The Jutta system allows generating and caching a highly optimized code fragment for a given joinpoint. This code fragment directly executes the appropriate advices on the applicable hooks, employing the sequence defined in the connector. As such, the system avoids iterating all connectors in order to retrieve the applicable aspectual behavior. Furthermore, rearranging the sequence of all applicable hooks for different advice types in order to implement precedence strategies, is also avoided. Listing 14 illustrates the simplified Java counterpart of a cached combined aspectual behavior fragment. This fragment implements the execution of four advice methods on three different hooks.

The current JAsCo implementation employs the Javassist byte-code manipulation library in order to generate a combined hook behavior code fragment. Using Javassist, a Java byte code class representation is generated on the fly, without requiring an additional compilation step. The overhead of generating a combined hook behavior code fragment is around 10ms on our test system[6]. The optimized code fragment is only generated when the joinpoint is encountered for the first time. Hence, no overhead is experience for joinpoints that are not executed. The Jutta system stores all code fragments generated for a given hook combination. As such, when the same hook combination is applicable to different joinpoints, which is typically the case, the overhead for generating the combined

[6] Pentium4 2 GHz, 256MB RAM, Ubuntu Linux 5.04, Java 1.5.0 update 1

hook behavior code fragment is avoided. In addition, the Jutta system includes a set of pre-defined typical combined aspectual behaviors. For those combined aspectual behaviors, the generation overhead is also avoided.

The JAsCo approach is a dynamic AOP approach. As such, the cached behavior for a given joinpoint might become invalid. This event occurs 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 Jutta system is able to cope with these issues.

Caching combined aspect behavior is not always achievable, as sometimes, the applicability of a hook needs 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 is ever applicable by examining the call graph of an application. This technique is elucidated in [24].

3.2 HotSwap

The Jutta system allows optimizing the aspect interpretation part of the JAsCo run-time environment. The interception part however, is still very slow. Inserting traps at all methods causes a performance overhead, even if no aspects are applicable. In order to optimize this interception system, we present a custom-made byte-code instrumentation framework, called HotSwap. This framework 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. JAsCo HotSwap has two different implementations which depend 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.

Both libraries make sure that the byte-code replacement does not leave the application in an inconsistent state: methods that are being executed at the moment of a replacement request keep employing the old byte-code and only for new invocations, the newly introduced byte-code is used. The byte-code manipulation themselves are again implemented using the Javassist library. Both JDI and JPLIS do not allow to alter the schema of a class and only make it possible to re-implement method bodies. As a result, the resulting application's state remains consistent because it is impossible to e.g. remove data fields.

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, the applicable methods are hot-swapped at run-time with a trapped equivalent. 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 remain untouched. Likewise, the original method byte code is reinstalled when an aspect is removed again and if no other aspects are applicable on the method at hand.

The JAsCo HotSwap system does not exclude the regular preprocessing approach for installing traps. Classes that are already equipped with traps using the preprocessor are never altered. As such, when certain classes are always affected by aspects, they can be preprocessed to avoid the HotSwap overhead at run-time. Furthermore, on platforms where no HotSwap compliant virtual machine is available, like e.g. embedded devices, the preprocessing approach can still be employed.

3.3 Towards a genuine run-time weaver

By combining the JAsCo HotSwap and Jutta systems, a genuine run-time weaver can be realized. Instead of inserting traps, the Jutta optimized code fragments are directly injected into the target joinpoints. The JAsCo run-time weaver thus further optimizes the advice execution time as the indirection to the Jutta code fragments is avoided. Furthermore, the run-time weaver generates a unique code fragment for each joinpoint shadow instead of a shared code fragment for several joinpoints in case of Jutta. This allows to generate a very specific joinpoint implementation that can e.g. use the concrete argument types and thus avoids expensive casts. The precedence of the advices is computed as defined in the corresponding connectors and thus induces no additional run-time overhead. Listing 15 illustrates the Java counterpart of the inserted code at a certain method execution joinpoint. The advice invocations are directly inserted into the joinpoints.

The run-time weaver works on a per-class basis: it processes all advised joinpoints of a certain class at once. Because HotSwap does not allow schema changes, a helper class is generated that contains external information, like e.g. the joinpoints representation or the aspect instances in case of cachable factory instantiations. The main motivation for this strategy is that the HotSwap framework only allows to replace byte code of complete classes. Generating the new byte code for a given class at once thus helps to reduce the HotSwap overhead.

Implementing Joinpoint Contextual Access One major optimization 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. Most AOP implementations still capture all actual arguments in an array, which is a very expensive operation, even though they are not required. Because JAsCo has its own aspect compiler, analyzing the required context for each advice is possible. If an aspect only requires

```
1  public String myMethod(int arg1, Vector arg2) {
2      IJoinPoint jp = JPContainer.initJoinpoint73637(this);
3      MyHook hook1 = MyConnector.hook35;
4      MyOtherHook hook2 = AspectContainer536.hook40;
5  
6      hook1.before();
7      String result = hook2.around(jp,arg1);
8      hook1.after(this,arg2);
9  
10     return result;
11 }
```

Listing 15. Simplified Java counterpart of the inserted code at a certain joinpoint

the method name, only that static information 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, while in some cases it only requires the method name. However, in many cases, this analysis allows for a major optimization. For example, in listing 15, the only reflective information required is the target object, so only this information is passed onto the object representing the current joinpoint (line 2).

In the JAsCo language, advices do not have to declare which arguments they require, as the arguments of abstract method parameters have a scope that extends all over the hook. Under the hood however, advices do have arguments, namely those that are detected to be effectively used by the advice. This is also reflected in listing 15 where the different advices are invoked by supplying only those arguments effectively required. The *thisJoinPoint* and *thisJoinPointObject* keywords are also handled as arguments and only supplied when they are effectively required. For example, the around advice of `hook2` possibly employs the *thisJoinPoint* keyword, so the object representing this joinpoint is passed on as an argument.

Implementing Around Advices Around advices are able to invoke the original behavior of the wrapped joinpoint or the behavior of the next around advice in the advice chain, by employing the `proceed` keyword. Originally, proceed is implemented by invoking a special callback object that maintains the list of remaining around advices and also allows to invoke the original behavior. This is however quite costly because, apart from the additional indirection, the next around advice needs to be popped from this list for each proceed invocation. Therefore, the JAsCo run-time weaver employs *around advice in-lining*, which is a technique that is first introduced by AspectJ's weaver [25]. In short, around advice in-lining generates a copy of every around advice at a joinpoint. All proceed statements in these copies are transformed into the direct invocation of either the next around advice or the original behavior, depending on the position of the around advice in the around advice chain.

Implementing Hook Instance Management Per default, hook instances are kept as *static final* members of the connector class representation generated by the JAsCo connector compiler. For instance, listing 15 retrieves the correct instance of the `MyHook` hook type using the `MyConnector` class. JAsCo however allows to use both predefined and custom aspect factories that generate new hook instances depending on specific joinpoint information. In case the factory is cachable (e.g. `perclass` or by not implementing the `DoNotCache` interface for a custom factory), the aspect instance is constant given the static joinpoint shadow and the factory result can thus be optimized. This optimization consists of the on-the-fly generation of an aspect container class that keeps the hook instances for specific joinpoints as *static final* members. For instance, the `MyOtherHook` hook type has been instantiated using the *perclass* keyword and is thus retrieved by accessing the `AspectContainer536` class which has been generated to keep all hook instances for all joinpoints of the current class.

In case a hook is instantiated using a non-cachable factory, there is no other option than invoking the factory's logic for retrieving the correct hook instance given a particular joinpoint. The *perinstance* and *perthread* predefined aspect factories are however optimized. The former is implemented by adding a field that contains the hook instance for a given object. As such, the *perinstance* factory only requires a single field access for retrieving the correct instance. The *perthread* factory is implemented using a *ThreadLocal* variable, which is a Java specific mechanism to create variables that can have a different value depending on the executing thread that accesses the variable.

Implementing Combination Strategies Likewise to aspect factories, combination strategies can be cachable or non cachable. A cachable combination strategy always returns the same result given a certain input set. This means that the result of a cachable combination strategy is fixed for every execution of a given joinpoint. The result of the combination strategy can thus be used for generating the weaved code fragment for the current joinpoint. As such, no additional run-time overhead is experienced for a cachable combination strategy. A non-cachable combination strategy has to be executed for every execution of a joinpoint. The JAsCo run-time weaver optimizes non-cachable combination strategies by internally representing the list of hooks as an array of booleans. Each element in the array represents whether a given hook should be triggered. When removing or adding[7] hooks from the list, the appropriate boolean is altered. After executing all combination strategies, all the advices execute conditionally depending on the corresponding value in the boolean array.

[7] Notice that in a combination strategy one can only add hooks that are possibly triggered at that joinpoint. Hooks that based on their pointcut definition can never be triggered at that joinpoint can also not be added by the combination strategy. This limitation allows to keep the length of the boolean array decidable at weave-time.

Implementing Stateful Aspects The JAsCo run-time weaver implements stateful aspects by simulating a Deterministic Finite Automaton for every hook instance. When a joinpoint is encountered, the automaton is used to query a) whether the hook is applicable and b) which advice should be executed. All hooks' advices execute conditionally depending on the query's result. The concrete advice to execute is determined by executing a `switch` statement over the possible transition identifiers. The automaton's internal state is updated simultaneously with the querying.

Instead of statically weaving code at any possible joinpoint that might be required for the execution of the stateful aspect, the aspect is only woven at the applicable joinpoints 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 joinpoints in which the aspect is no longer interested and reweaves it at the appropriate joinpoints. This realizes a real *jumping aspect* [9] that literally jumps from joinpoint(s) to joinpoint(s). As a result, no unnecessary woven code is left at inapplicable joinpoints. Also notice that when the aspect is no longer applicable because the triggered transition does not define destination transitions, it is completely unwoven and thus does not cause a performance overhead any longer. Previous work by Costanza [26] also motivates that aspects should be able to *vanish*.

Discussion The main drawback of the run-time weaver is the increased run-time overhead for adding and removing aspects. In the trapped approach, adding a new aspect does not require any HotSwap overhead whatsoever, if a trap is already placed. 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(IJoinPoint 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 second. JAsCo thus effectively combines and integrates three alternative aspect weavers.

By removing aspects dynamically, it is possible that the application is left in an inconsistent state. This can happen when e.g. the aspect alters a part of the base application that is erroneous when the aspect is not present anymore. Therefore, aspects are able to receive state property change events that notify the aspect instance when the instantiating connector is added, removed, enabled or disabled.

4 Performance Evaluation

In this section, we present the results of several benchmarks evaluating different facets of JAsCo run-time weaver's performance. If possible, we compare the results of JAsCo with several other AOP approaches. The following two sections present two micro-benchmarks that respectively measure the joinpoint execution time and weaving time. Section 4.3 discusses the performance of JAsCo in three realistic application scenarios. Finally, we present several benchmarks that measure the performance of JAsCo specific features.

4.1 AWBench run-time benchmarks

As a first benchmark, we employ the independent AWBench [27] benchmark. This benchmark is a project of the AspectWerkz team and is especially designed to compare the performance of AOP systems. AWBench is a micro benchmark and consists of thirteen 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 compare the performance of JAsCo with the following AOP approaches: AspectJ 1.2 [5], JBoss/AOP 1.0 [13], AspectWerkz 2.0 [28], Spring/AOP 1.1.1 [14]. All approaches, which the exception of AspectJ, allow some form of run-time aspect application and removal, although this is limited in most cases. 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 uses a traditional weaver that invasively weaves the aspects into the target classes at compile-time and as such does not allow dynamic aspect application and removal. The AspectWerkz technology is based on our Jutta system, which has been significantly improved by the AW team. For example, every joinpoint has its unique on-the-fly generated class which invokes the correct advices instead of a shared jutta class. By cleverly exploiting common JIT's in-lining strategies, AspectWerkz is able to reduce the joinpoint execution overhead significantly. AspectWerkz allows the application and removal of aspects at joinpoints where there are already aspects applied. Other joinpoints cannot be subject of dynamic aspect application or removal. JBoss/AOP uses an approach similar to the original JAsCo technology, namely inserting traps to all advised joinpoints. Contrary to JAsCo, the traps are installed at load-time and can never be removed. As such, at joinpoints where no traps are attached, dynamic aspect application is impossible. Spring/AOP is a proxy-based approach that employs the Java Dynamic Proxies feature to dynamically attach advices to objects. Similar to JAsCo, Spring/AOP fully supports dynamic addition and removal of aspects, even at previously unadvised joinpoints. However, Spring/AOP only

supports method execution joinpoints and endures a relatively large performance overhead because of the use of Dynamic Proxies.

Table 1 illustrates the results of running the AWBench using the introduced approaches on our test system[6]. The performance of JAsCo using traps combined with the Jutta system is also recorded. The performance of the deprecated JAsCo interpreter is also shown in order to indicate the cost of such an interpreted approach. In all benchmarks AspectJ, AspectWerkz and the novel JAsCo run-time weaver 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 Jutta) perform worse than weaving, but still execute considerably faster than the proxy-based approaches (Spring/AOP).

	JAsCo RT Weaver	**AspectJ**	**AspectWerkz**	**JBOSS/ AOP**	**Spring/ AOP**	**JAsCo Jutta**	**JAsCo Interpreter**
before	**15**	**17**	**15**	305	493	206	244×10^5
before refl stat	**10**	**11**	34	296	446	195	256×10^5
before refl dyn	**13**	94	116	298	455	195	255×10^5
before decl args	**11**	**12**	**10**	412	607	264	338×10^5
before decl args2	**11**	**12**	**10**	369	553	221	388×10^5
before decl all	**12**	**13**	**10**	433	620	262	336×10^5
around	**10**	**11**	152	291	431	211	246×10^5
around refl stat	**10**	**12**	175	279	433	209	258×10^5
around refl dyn	**11**	97	155	296	419	207	253×10^5
after returning	**10**	**11**	**12**	299	437	170	363×10^5
after throwing	5221	**5159**	5655	10782	*n/a*	8697	295×10^5
before+after	**16**	23	25	367	681	171	270×10^5
aroundx2	**17**	82	189	551	697	344	364×10^5

Table 1. AWBench benchmark results. The values show the overhead per joinpoint in ns. Differences of five nanoseconds or less are not significant because they are too close to the duration of a clock cycle. Static reflective context (*refl stat*) access consists of reflectively accessing information (method name in this case) that does not have to be computed at run-time, but remains constant for every execution of the joinpoint shadow. Dynamic reflective context access (*refl dyn*) consists of fetching dynamic contextual values (target object in this case) of the joinpoint reflectively using the `thisJoinPoint` keyword. Declaratively fetching joinpoint contextual information (*decl **) means explicitly defining the dependence on that information in the pointcut description

For before advices where the run-time context is fetched declaratively, AspectJ, AspectWerkz and JAsCo perform equally well. All three optimize the joinpoint 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, the results of 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.

4.2 Weave-time benchmarks

The AWBench benchmark allows to precisely measure the overhead for advice execution per joinpoint. The weaving cost itself is not included in the performance results of the AWBench project. In a Java context, the run-time execution cost is generally the most important performance criterion. Java is mainly used for server applications where the virtual machine ideally runs for a very long period. However, in some cases, the time required for weaving aspects, cannot be ignored. For example, when developing and testing applications, the virtual machine is typically restarted quite often. Furthermore, because JAsCo employs run-time instrumentation, adding and removing aspects frequently might also cause an unacceptable overhead for the complete application. Therefore, we employ a custom benchmark application to measure the run-time weaving overhead. The benchmarks measures the cost for weaving 1000 method execution joinpoints and executing all these methods once. Executing the methods is mandatory because some AOP approaches (like AspectWerkz and JAsCo Jutta) use a form of lazy weaving where parts of the weaving process are postponed until the first execution of the joinpoint shadow.

	JAsCo RT Weaver	**Aspect-Werkz**	**JBOSS/ AOP**	**SPRING/ AOP**	**JAsCo Jutta**
before advice	**15**	**15**	36	26	19
around advice	21	**15**	36	26	19

Table 2. Time required for weaving in ms per joinpoint

Table 2 illustrates the results for the weaving benchmarks on a per joinpoint basis. We compare the weaving cost of the JAsCo run-time weaver with three other approaches that only weave aspects while running the application. As explained before, AspectJ weaves aspects at compile-time and as such does not endure a run-time performance overhead for the weaving. The JAsCo run-time weaver performs equally well as AspectWerkz in case of before advices. The overhead for around advice in-lining as explained in section 3.3 causes an additional weaving cost of 40 % in comparison to the before advice. Contrary to what one might expect, the cost of weaving a before advice in the trapped JAsCo/Jutta approach is higher than invasive weaving. This is explained by the fact that the trapped approach inserts a trap independent of the concrete aspects applied. As such, for every trap, a callback class for implementing the around advice closure [25] has to be generated. The run-time weaver generates a code fragment unique

for the joinpoint and given advices and is able to omit the generation of this callback object in case no around advices are present.

4.3 Realistic Application Benchmarks

In this section, we assess the performance of the JAsCo run-time weaver in a realistic application setting rather than micro benchmarks as performed in the previous sections. We use two benchmark applications, *PacoSuite* a component composition environment [29] and *sim* a discrete event simulator for certificate revocation simulation [30]. The *PacoSuite* benchmark loads and model-checks a component composition. One before advice is applied that updates a counter for every method execution. The *sim* benchmark is presented in [31] and defines an aspect that checks whether the returned output is null for every method execution. The *sim* benchmark consists of 20 classes while the PacoSuite benchmark consists of over 500 classes. As a third benchmark, we reuse the *sim* application but apply five after advices to all public method execution joinpoints.

	Time s				**Memory kb**			
	Unad-viced	**JAsCo RT**	**AJ**	**AW**	**Unad-viced**	**JAsCo RT**	**AJ**	**AW**
Sim1400	6.244	7.705	**6.376**	8.804	14012	**7367**	14078	8713
Sim3500	122.1	132.8	**130.9**	134.4	17698	**11015**	17825	12207
Paco10	14.58	24.41	**14.89**	28.74	18598	30138	**18585**	37673
Paco100	107.4	110.7	**108.2**	115.2	19178	32283	**19495**	43686
Paco1000	9763	9767	**9764**	9770	21377	36108	**21213**	48452
5Sim1400	6.244	8.588	**7.603**	10.91	14012	**7558**	14091	9423
5Sim3500	122.1	**151.2**	152.6	163.2	17698	**11166**	17697	12744

Table 3. Two realistic application scenarios run using JAsCo, AspectJ and AspectWerkz. Sim defines a checking aspect for null pointer returns, PacoSuite defines a counter aspect using a before advice and 5Sim defines five after advices. The number suffix of each benchmark name refers to the number of iterations in case of PacoSuite and the startup parameters in case of sim. The resulting values are the total execution time in seconds and the maximum allocated memory in kilobytes

The overhead for weaving the classes at run-time is high in case the application runs for only a short period of time. Both *Paco10* and *Sim1400* endure an overhead of respectively 20 % and 60 % over the execution time of AspectJ. The difference is explained by the small code base of *sim* in comparison to *PacoSuite*. As such, weaving *PacoSuite* is a lot more costly than weaving the *sim* application. When the application runs for a longer time, the difference in performance becomes a lot smaller. In fact, for *Paco500* and *Sim3500*, JAsCo executes less than two percent slower than the equivalent AspectJ application. When the application is run for an even longer time like the *Paco1000* benchmark, the difference between AspectJ, JAsCo and AspectWerkz is negligible.

In case of the *5sim* benchmark, which applies five after advices for every method execution, JAsCo is able to catch up with AspectJ when the application is run long enough. This is also confirmed by the AWBench results when several advices are applicable to the same joinpoint (*before+after* and *around*$\times$2), where

JAsCo is able to improve over AspectJ's joinpoint execution time. At first sight, five advices per public method execution joinpoint might be unrealistic, but this might very well be the case in an application that relies heavily on AOP for several crosscutting concerns like Transaction Management, Security, Tracing, etc.... In any case, the JAsCo run-time weaver is able to compete with AspectJ performance-wise as soon as the application runs long enough (which is typically the case in a Java application server scenario) and thus our goal has been realized: the JAsCo run-time weaver combines the flexibility of run-time weaving with the performance of a static weaver.

Contrary to what one might expect, JAsCo requires less memory for the *Sim* benchmarks than running the benchmark with AspectJ. In fact, running the benchmark with both JAsCo and AspectWerkz requires even less memory than running the benchmark without AOP. This strange result is probably explained by the use of the JPLIS VM plugin, which apparently causes the VM to optimize memory usage in case of small applications. In the larger *Paco* benchmark application, the JAsCo run-time infrastructure requires more memory (around 50%) than the equivalent AspectJ program. This overhead is caused by the additional run-time infrastructure for weaving and unweaving joinpoints at run-time. Notice that JAsCo is not yet optimized memory-wise and we expect that we can reduce this overhead significantly. However, due to the required weaving infrastructure, JAsCo normally requires more memory than compile-time weavers like AspectJ.

4.4 JAsCo specific benchmarks

We also conduct several benchmarks that measure the cost for specific JAsCo features in order to motivate these additional JAsCo language features from a performance point of view. The specific JAsCo language features that are benchmarked are: predefined and custom aspect factories, combination strategies and stateful aspects.

Aspect Factories Table 4 illustrates the results of a custom benchmark that deploys aspects using aspect factories. The aspect defines one simple before advice. Some of the factories are also supported in other approaches, so we included the performance of these approaches as well for comparison. The result of `perclass` is constant for every shadow joinpoint, so no performance overhead is visible. Likewise, custom factories that are cachable induce no additional overhead. JAsCo executes `perinstance` significantly faster than e.g. AspectWerkz, which requires a hash-table access for implementing this feature.

Custom non-cachable factories require a larger overhead due to a) the computation cost for fetching the correct instance (a hash-table access in this case) and b) the overhead for fetching all joinpoint contextual information because the JAsCo system does not know which information the factory might require. The latter overhead might be reduced by implementing a lower-level factory interface, which is still completely modular and does not require adaptations to the

JAsCo weavers. However, implementing a low-level factory requires knowledge of the byte-code manipulation library Javassist and is thus not so user-friendly as the standard factory implementation.

Perall induces a very large performance overhead for instantiating and initializing a hook for every joinpoint execution. As such, it seems that in most cases it is better to use a workaround in order to avoid using this factory.

	JAsCo RT Weaver	AspectJ	AspectWerkz	JBoss/AOP
simple before	**11**	**10**	**10**	301
perclass	**11**	*n/a*	**11**	354
perinstance	**22**	77	137	467
perthread	88	*n/a*	*n/a*	*n/a*
perall	38705	*n/a*	*n/a*	*n/a*
custom cachable	11	*n/a*	*n/a*	*n/a*
custom non-cachable	311 (181)	*n/a*	*n/a*	*n/a*

Table 4. Aspect factory benchmarks measures the overhead per joinpoint in ns

before	+ Cachable combination strategy	+ Non-cachable combination strategy
11	11	109

Table 5. Combination strategy benchmark measures the overhead per joinpoint in ns

Combination Strategies Table 5 illustrates the result on the joinpoint execution time when combination strategies are present. The benchmark executes one before advice that has to be filtered by one combination strategy. When the combination strategy is cachable, the result is determined at weaving time, and no overhead with respect to the before advice execution in AWBench can be observed. In case of a non-cachable combination strategy, the joinpoint executes about 10 times slower. This is the overhead required for initializing the internal boolean array representation, executing the combination strategy and conditionally executing the before advice.

manual stateful before	stateful 1	stateful 2	stateful 3	stateful 4	stateful 5
27	136	135	141	143	147

Table 6. Stateful Aspects benchmark measures the overhead per joinpoint in ns. Depending of the number of outgoing transitions, the overhead increases slightly

Stateful Aspects Stateful aspects are implemented by simulating a deterministic finite automaton at run-time. The automaton is implemented using a dictionary, which means that finding the matching transition for a given state costs O(log n) with n the number of outgoing transitions at that state. Table 6 shows the performance for executing a stateful aspect that defines a before

advice for every transition. Depending on the number of outgoing transitions (i.e. transitions to follow next after firing a transition pointcut), the cost for executing a stateful aspect's advice increases slightly.

In comparison with a highly optimal manual hard-coded implementation, the JAsCo implementation executes around six times slower. This is due to the explicit simulation of a state machine at run-time. Realizing a performance similar to the hard-coded implementation is feasible and subject for future work. Notice however, that even the non-optimal JAsCo stateful aspect implementation executes a lot faster than a simple before advice in e.g. JBoss/AOP.

5 Related Work

Nowadays, several both static and dynamic aspect-oriented technologies are available. When JAsCo was originally conceived, AOP was less popular and only a handful of approaches existed. The JAsCo language is inspired by two of those pioneer AOP approaches: AspectJ [5] and Aspectual Components [7]. Aspectual Components motivates that in order to increase the reusability of aspects, a separate deployment phase is required. Therefore, Aspectual Components introduces the connector concept also known in JAsCo. JAsCo combines this idea with the expressive pointcut language and the asymmetric base-aspect model of AspectJ. Furthermore, JAsCo extends both approaches, with at the one hand more expressive connectors that support precedence and combination strategies and at the other hand support for dynamic pointcut conditions such as stateful pointcuts. Also on the technical level, JAsCo is quite different supporting efficient run-time weaving where both the Aspectual Components model and AspectJ make use of static byte code weaving.

Caesar [32] describes aspects in terms of an Aspect Collaboration Interfaces (ACI). Each concrete aspect needs to implement the required methods specified by its corresponding ACI. Aspect bindings connect the aspect implementations to different concrete component 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. We are working on an extension of the JAsCo language that allows to refine refinable methods based on the joinpoint's target type. These type-specific refinements are completely modular and are lately and automatically bound.

Event-based aspect-oriented programming (EAOP) allows specifying crosscutting concerns by employing event patterns which are described using a formal language [33]. Because of this formal model, advanced detection and resolution of aspect interactions becomes feasible. At 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. EAOP however relies on the availabil-

ity of the source-code of the complete application as it inserts traps by using source-code transformations.

In the benchmark section, AspectWerkz, JBoss/AOP and Spring/AOP are introduced as current practice aspect-oriented approaches. AspectWerkz is a stand-alone framework while JBoss and Spring are J2EE application servers with AOP support. These three approaches describe reusable aspects as regular Java classes. Aspects are deployed within the system using a deployment descriptor which is expressed using XML. All three approaches however lack support for aspect instantiation control, stateful pointcuts and aspect combination control. Also, JBoss/AOP and Spring/AOP do not support strong aspectual typing, i.e. arguments of joinpoints for instance need to be fetched from an array and casted to the appropriate types at run-time.

Steamloom [34] is another dynamic AOP approach that supports the run-time weaving of aspects. Similar to PROSE2 [35], 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. Steamloom is implemented as an extension of IBM's Jikes Research Virtual Machine [36] and extends the Java byte code language with AOP specific instructions. This strategy has two main benefits. First of all, traditional weaving is not really necessary anymore, at most some additional instructions have to be inserted. This makes the aspect application and removal process a lot easier to implement. A second advantage is that these specific instructions can be implemented very efficiently at the VM level. The main drawback however is that the approach loses compliance with the Java standards by extending the Java byte code language. As such, a custom virtual machine needs to be employed. The JAsCo run-time weaver is implemented as a plugin into the VM and uses only standardized interfaces and is thus portable over all recent JDK compliant virtual machines.

JAC, DAOP and OIF are three approaches that target component-based systems and allow dynamic AOP. JAC [37] employs load-time trap insertion, which are similar to the traps inserted by the JAsCo preprocessor. As a result, JAC suffers from high run-time advice execution cost as well. The Dynamic Aspect-Oriented Platform (DAOP) [38] is another approach that targets component-based systems. DAOP introduces a distributed platform, where a middleware layer stores all composition information. In this respect, it is similar to the connector registry employed in JAsCo. Filman [39] proposes dynamic injectors in order to introduce aspects within a component-based application. These dynamic injectors are incorporated into the OIF (Object Infrastructure Framework), a CORBA centered aspect-oriented system for distributed applications. At the implementation level, a wrapping approach is employed for injecting the logic of an aspect within a component communication channel.

Wool [40] is another dynamic AOP approach that supports two types of dynamic weaving strategies. Similar to PROSE [41], 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 the Java Debugging Interface (JDI). Both the Wool language and weavers are however quite basic and only support simple advices like a before/after advice. 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 heuristic function can be specified.

Strategic programming (SP) [42] is a generic programming idiom for processing compound data, such as parse trees of programming languages. It was initiated in the context of term rewriting (using Stratego [43]), but has been transposed to other programming paradigms such as functional programming (based on Strafunski [44]) and object-oriented programming [45]. Similar to JAsCo's combination strategies, SP allows programmer-definable combinators. The definition of these combinations relies on primitives and lower-level combinators. As such, by building on a small suite of combinators, a wide and expressive variety of combinations can be defined in a declarative way. The main difference with JAsCo combination strategies is that SP combinators are specified declaratively, which has several advantages regarding understandability, automatic optimizations and analysis. In JAsCo however, we explicitly choose for an imperative approach as this allows employing the full expressiveness of Java.

Walker et al. introduce *declarative event patterns* (DEPs) [46] 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. The current proposal also suffers from performance issues due to the grammar-based approach.

Recently, another implementation of protocol-matching pointcuts for AspectJ has been proposed [47] in the context of the AspectBench Compiler framework. The abc proposal is similar to DEPs, but protocols are specified as regular expressions of expected events. Additionally, this implementation emphasizes the importance of crosscut variables (as introduced by Douence *et al.* in [17]). These are variables that can be bound and matched against as the protocol progresses. Abc does however neither support attaching advices to specific transitions during the protocol.

6 Conclusions and Future Work

In this paper, we introduce the JAsCo approach that aims at tackling several problems encountered in current practice AOP. Aspects described using JAsCo are context-independent and first-class entities while still providing aspectual type safety. A separate connector construct is used to deploy an aspect onto target components. Connectors allow explicit control over aspect instantiation and initialization. JAsCo partly addresses the feature interaction problem by allowing the ordering of conflicting aspect behaviors and by introducing explicit and reusable combination strategies. Although JAsCo combination strategies are very expressive, they are quite ad-hoc and often complicated. We are currently working on a dedicated aspect combination language that makes the combination itself more explicit.

To make the JAsCo language operational, we propose the Jutta aspectual just-in-time compiler and the HotSwap run-time byte-code instrumentation framework. By combining both the HotSwap and Jutta technologies, a run-time weaver is realized. As such, JAsCo supports three types of weaving: pre-processed trap insertion, run-time trap insertion and run-time aspect weaving. The three weaving alternatives can be used simultaneously and thus allow for detailed performance fine-tuning.

Although JAsCo is currently still under development, several research and industrial projects are currently employing JAsCo. In the context of the MOSAIC project in collaboration with a large telecom company, a Web Services Management Layer [48] (WSML) is developed. The WSML captures several crosscutting concerns such as security, load balancing, caching, WS selection, pre-fetching and fault tolerance as reusable dynamic JAsCo aspect beans. In the MOSAIC follow-up project, the deployment of JAsCo is to be evaluated in the context of asynchronous telecommunications platforms, such as supported by JAIN SLEE [49]. The ASPECTLAB project in collaboration with two other universities and twelve Flemish based companies (ranging from SMEs to multinationals and large governmental departments) evaluates JAsCo among other technologies for several industrial pilots. JAsCo is also used for teaching at several universities world-wide. The fact that JAsCo stays very close to the original Java syntax, while supporting a wide range of AOP concepts, makes it an attractive language for teaching AOP.

At a more conceptual level, we are investigating the advantages of a full unification between aspects and components. Most AOP approaches available today propose an asymmetric aspect model where aspects are treated and implemented as different kind of entities. In our opinion, a symmetric model is more appropriate as only the interaction of an aspect with other components requires special treatment. Therefore, we propose a unified component framework, called FuseJ [50], which provides an expressive component composition mechanism that allows describing aspect-oriented as well as component-based interactions between components.

References

1. McIlroy, M.D.: “Mass produced” software components. In Naur, P., Randell, B., eds.: Software Engineering, Brussels, Scientific Affairs Division, NATO (1969) 138–155 Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.
2. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. 1st edn. Addison Wesley, Reading, Massachusetts, USA (1998)
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: 11th Europeen Conf. Object-Oriented Programming. Volume 1241 of LNCS., Springer Verlag (1997) 220–242
4. Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the Hyperspace approach. In Akşit, M., ed.: Software Architectures and Component Technology: The State of the Art in Research and Practice, Kluwer Academic Publishers (2001)
5. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: Proc. ECOOP 2001, LNCS 2072, Berlin, Springer-Verlag (2001) 327–353
6. Suvée, D., Vanderperren, W.: JAsCo: An aspect-oriented approach tailored for component based software development. In Akşit, M., ed.: Proc. 2nd Int’ Conf. on Aspect-Oriented Software Development (AOSD-2003), ACM Press (2003) 21–29
7. Lieberherr, K., Lorenz, D., Mezini, M.: Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA (1999)
8. Pulvermüller, E., Speck, A., Coplien, J., D’Hondt, M., De Meuter, W.: Feature interaction in composed systems. In: ECOOP ’01: Proceedings of the Workshops on Object-Oriented Technology, London, UK, Springer-Verlag (2002) 86–97
9. Brichau, J., De Meuter, W., De Volder, K.: Jumping aspects. In Lopes, C., Bergmans, L., D’Hondt, M., Tarr, P., eds.: Workshop on Aspects and Dimensions of Concerns (ECOOP 2000). (2000)
10. Filman, R., Haupt, M., Mehner, K., Mezini, M., eds.: DAW: Dynamic Aspects Workshop. In Filman, R., Haupt, M., Mehner, K., Mezini, M., eds.: DAW: Dynamic Aspects Workshop. (2004)
11. Filman, R.E., Haupt, M., Hirschfeld, R., eds.: Dynamic Aspects Workshop. In Filman, R.E., Haupt, M., Hirschfeld, R., eds.: Dynamic Aspects Workshop. (2005)
12. De Win, B., Vanhaute, B., De Decker, B.: How aspect-oriented programming can help to build secure software. Informatica **26** (2001) 141–149
13. Burke, B., et al.: JBoss Aspect-Oriented Programming. Home page at `http://www.jboss.org/products/aop` (2004)
14. Johnson, R., et al.: Spring Java/J2EE Application Framework, Reference Documentation. (2004) Available at `http://www.springframework.org/docs/spring-reference.pdf`.
15. Bouma, L., et al., eds.: Proceedigns of the Feature Interaction Workshops in Telecommunications and Software Systems, IOS Press (2004)
16. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: 1st Conf. Generative Programming and Component Engineering. Volume 2487 of lncs., Berlin, Springer-Verlag (2002) 173–188
17. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In Lieberherr, K., ed.: Proc. 3rd Int’ Conf. on Aspect-Oriented Software Development (AOSD-2004), ACM Press (2004) 141–150

18. Douence, R., Fradet, P., Südholt, M.: Trace-based aspects. In Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2005) 201–217
19. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2005) 21–35
20. Lopez-Herrejon, R., Batory, D., Cook, W.: Evaluating support for features in advanced modularization technologies. In: ECOOP 2005—Object-Oriented Programming, 19th European Conference. (2005) (to appear).
21. Chiba, S., Nishizawa, M.: An easy-to-use toolkit for efficient Java bytecode translators. In: GPCE '03: Proceedings of the second international conference on Generative programming and component engineering, New York, NY, USA, Springer-Verlag New York, Inc. (2003) 364–376
22. Verspecht, D., Vanderperren, W., Suvée, D., Jonckers, V.: JAsCo.NET: Capturing Crosscutting Concerns in .NET Web Services. In: NCWS'03: Proceedings of the second Nordic Conference on Web Services. (2003)
23. Deutsch, L.P., Schiffman, A.M.: Efficient implementation of the smalltalk-80 system. In: POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM Press (1984) 297–302
24. Sereni, D., de Moor, O.: Static analysis of aspects. In Akşit, M., ed.: Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003), ACM Press (2003) 30–39
25. Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In Lieberherr, K., ed.: Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004), ACM Press (2004) 26–35
26. Costanza, P.: Vanishing aspects. In Tarr, P., Bergmans, L., Griss, M., Ossher, H., eds.: Workshop on Advanced Separation of Concerns (OOPSLA 2000). (2000)
27. Bonér, J., Vasseur, A.: AspectWerkz AWBench project. Home page at `http://docs.codehaus.org/display/AW/AOP+Benchmark` (2004)
28. Bonér, J., Vasseur, A.: AspectWerkz: simple, high-performant, dynamic, lightweight and powerful AOP for Java. Home page at `http://aspectwerkz.codehaus.org/` (2004)
29. Wydaeghe, B., Vanderperren, W.: Visual component composition using composition patterns. In: Proceedings of Tools International Conference, IEEE Computer Society, Santa Barbara, USA (2001)
30. Arnes, A.: A discrete event simulator for certificate revocation simulation. Home page at `http://www.pvv.ntnu.no/~andrearn/certrev/sim.html` (2000)
31. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotk, J., Lhotk, O., Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Optimising aspectj. In: Proceedings of Programming Language Design and Implementation (PLDI) International Conference, Chicago, USA (2005)
32. Ostermann, K., Mezini, M.: Conquering aspects with Caesar. In Akşit, M., ed.: Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003), ACM Press (2003) 90–99
33. Douence, R., Südholt, M.: A model and a tool for event-based aspect-oriented programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes (2002)
34. Bockisch, C., Haupt, M., Mezini, M., Ostermann, K.: Virtual machine support for dynamic join points. In Lieberherr, K., ed.: Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004), ACM Press (2004) 83–92

35. Popovici, A., Alonso, G., Gross, T.: Just in time aspects. In Akşit, M., ed.: Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003), ACM Press (2003) 100–109
36. IBM Research: Jikes Research Virtual Machine. Home page at `http://jikesrvm.sourceforge.net/` (2001)
37. Pawlak, R., Seinturier, L., Duchien, L., Martelli, L., Legond-Aubry, F., Florin, G.: Aspect-oriented software development with Java Aspect Components. In Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2005) 343–369
38. Pinto, M., Fuentes, L., Fayad, M., Troya, J.M.: Separation of coordination in a dynamic aspect oriented framework. In Kiczales, G., ed.: Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002), ACM Press (2002) 134–140
39. Filman, R.E., Barrett, S., Lee, D.D., Linden, T.: Inserting ilities by controlling communications. Comm. ACM **45** (2002) 116–122
40. Sato, Y., Chiba, S., Tatsubori, M.: A selective, just-in-time aspect weaver. In: GPCE '03: Proceedings of the second international conference on Generative programming and component engineering, New York, NY, USA, Springer-Verlag New York, Inc. (2003) 189–208
41. Popovici, A., Gross, T., Alonso, G.: Dynamic weaving for aspect-oriented programming. In Kiczales, G., ed.: Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002), ACM Press (2002) 141–147
42. Lämmel, R.: Typed Generic Traversal With Term Rewriting Strategies. Journal of Logic and Algebraic Programming **54** (2003) Also available as arXiv technical report cs.PL/0205018.
43. Visser, E., Benaissa, Z.e.A., Tolmach, A.: Building program optimizers with rewriting strategies. In: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98), ACM Press (1998) 13–26
44. Lämmel, R., Visser, J.: Typed Combinators for Generic Traversal. In: Proc. Practical Aspects of Declarative Programming PADL 2002. Volume 2257 of LNCS., Springer-Verlag (2002) 137–154
45. Visser, J.: Visitor combination and traversal control. In: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2001) 270–282
46. Walker, R., Viggers, K.: Implementing protocols via declarative event patterns. In: Foundations of Software Engineering (FSE), ACM (2004) 159–169
47. Allan, C., et al.: Adding trace matching to AspectJ. Technical Report abc-2005-1, Oxford University (2005)
48. Verheecke, B., Cibran, M., Vanderperren, W., Suvee, D., Jonckers, V.: AOP for Dynamic Configuration and Management of Web Services. International Journal of Web Services Research (JWSR) **1(3)** (2004)
49. O'Doherty, P.: JAIN SLEE principles. Available at `http://java.sun.com/products/jain/article_slee_principles.html` (2003)
50. Suvée, D., Vanderperren, W., Wagelaar, D., Jonckers, V.: There are no aspects. In Aßmann, U., Pop, A., eds.: Software Composition Workshop (ETAPS). (2004) 142–162