Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving

Stefan Hanenberg
University of Duisburg-Essen
Department for Computer Science
and Business Information Systems
D-45117 Essen, Germany
shanenbe@cs.uni-essen.de

Robert Hirschfeld
DoCoMo Communications
Laboratories Europe
Future Networking Lab
D-80687 Munich, Germany
hirschfeld@docomolab-euro.com

Rainer Unland
University of Duisburg-Essen
Department for Computer Science
and Business Information Systems
D-45117 Essen, Germany
unlandr@cs.uni-essen.de

ABSTRACT

Weaving is one of the fundamental mechanisms of aspect-oriented systems. A weaver composes different aspects with the base system by determining and adapting all parts where aspect specific elements are needed eventually. At runtime, time-consuming join point checks are necessary to determine if at a certain join point aspect-specific code needs to be executed. Current technologies enforce such checks even in locations that only temporarily or under restrictive conditions (or even never) execute aspect-specific code. In more complex applications, a large number of these checks fail and just cause a substantial runtime overhead without contributing to the system's overall behavior. The main reason for this flaw is *complete weaving*, the way how aspects are woven to an application using current technologies. In this paper we discuss the problem of unnecessary join point checks caused by complete weaving. We introduce *morphing aspects* – incompletely woven aspects in combination with *continuous weaving* – to overcome the problem of futile join point checks.

1. INTRODUCTION

Aspect-Oriented Programming [17, 20] deals with code fragments which logically belong to certain concerns but which cannot be modularized due to limited composition mechanisms of underlying programming languages and environments. The resulting code is *tangled* and *scattered*. Concerns that cause such tangling are called *crosscutting concerns*. Aspect-orientation is about modularizing crosscutting concerns into distinct modules, called *aspects*.

The mechanism for integrating aspect modules with an application is called *weaving*. A weaver is responsible for adding all aspects to the application. In order to specify such integration, aspect-orientation makes use of a concept called *join point*. In [17], join points are introduced as principled points in the execution of a program. A typical example of a join point is a method call.

AOSD 04, March 2004, Lancaster UK.

Conventionally, the weaving process is started by the developer at a certain point in time and for a number of aspects to be integrated. Thereto, the weaver determines and adapts all locations in the base system which represent join points at runtime where potentially or for sure aspect-specific code needs to be executed. In [22], locations that represent join points during runtime are called *join point shadows*. In the following we refer to shadows whose join points always lead to an execution of aspect-specific code as *unconditional join point shadows*, and those whose join points lead only under some circumstances to aspect-specific behavior as *conditional join point shadows*. For conditional shadows the weaver adds runtime checks determining whether or not aspect-specific code needs to be executed (we refer to these runtime checks as *join point checks*). If such a check succeeds, the aspect-specific code (the *advice* code according to AspectJ terminology [18]) is executed.

One property of this conventional approach to weaving is that the set of join point shadows associated with a woven aspect remains the same for the aspect's lifetime. In the following we refer to this kind of weaving as *complete weaving*. Complete weaving is a process which determines and adapts all join point shadows including the creation of corresponding join point checks upfront and in advance. After weaving, all shadows in the application where aspect-specific code might be executed are adapted. Consequently, the set of join point shadows associated to an aspect is fixed and does not change at runtime. Aspect-oriented systems like AspectJ [18], Hyper/J [24] and Sally [12] that provide *pure static weaving*, i.e. weaving at compile time, necessarily need to perform a complete weaving since all join point shadows to be adapted have to be determined at a certain point in time (at compile time[1]).

In more complex applications complete weaving can lead to a huge number of adapted join point shadows whose join point checks fail and just produce runtime overhead. Especially shadows with join points that rarely trigger the execution of aspect-specific code in the execution of the program are useless

[1] It should already be emphasized here that the term complete weaving is not equivalent to static weaving. A form of complete weaving also occurs in systems that provide dynamic weaving. This will be discussed in more detail in section 6.

time-consumers because most of the time the corresponding join point checks do not succeed and with that do not invoke an aspect's advice. In the worst case an aspect adapts a large number of join point shadows that never invoke an aspect's advice. In such cases the adapted shadows cause runtime overhead without ever accomplishing any benefit at all.

A large number of conditional shadows that rarely or never lead to an advice's execution are often not tolerable, especially not in performance critical parts of the system. Thus, it is desirable to reduce the number of conditional join point shadows as much as possible to reduce the number of failing and with that unnecessary join point checks.

In order to reduce the number of conditional join point shadows we introduce the concept of *morphing aspects* which are incompletely woven aspects in combination with *continuous weaving*, an extension of *dynamic weaving* [16, 26]. In the next section we provide two typical examples of aspects based on complete weaving that illustrate the necessity of handling the problem of conditional shadows whose join points rarely or never execute an aspect's advice. In section 3, we introduce the concept of morphing aspects. We discuss dependency relationships among join points and describe how they can be used to adapt join point shadows at a later point in time. In section 4, we discuss implementation issues of morphing aspects by proposing an implementation in *AspectS* [16]. We give an overview of some experiments with morphing aspects in section 5. After comparing morphing aspects to related work in section 6 we discuss and conclude our paper in section 7.

2. EXAMPLES

According to for example [16, 20] and [11, 30, 32] *tracing* and *subject-observer* implementations are well-known and accepted candidates for discussions and illustrations of aspect-oriented programming. Because of its popularity we use AspectJ in those examples to better illustrate the problems associated with complete weaving[2].

2.1 Tracing

A woven tracing aspect captures messages sent to or from particular objects, e.g. to a log file. Usually, developers want to trace the control flow starting at a certain point in the execution of a program. For example, a developer wants to capture the behavior of critical modules in order to analyze their behavior either later or right at runtime.

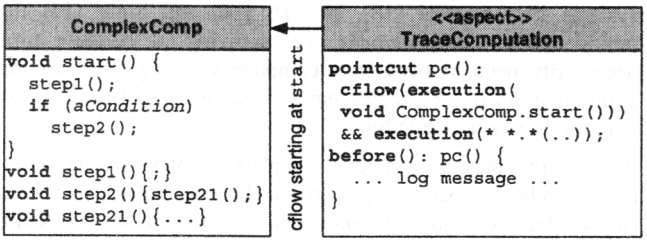

Figure 1. Tracing aspect in AspectJ logging methods in the control flow starting at method `start` in `ComplexComp`.

A typical approach to implementing tracing in AspectJ is to use the `cflow` pointcut designator [18]. Figure 1 shows the corresponding code in AspectJ where a tracing aspect `TraceComputation` logs all messages once the control flow passes method `start` in class `ComplexComp` which starts a complex computation[3]. One advantage of this implementation is the declarative pointcut definition that describes all join points where the tracing aspect needs to execute some advice. Hence, developers do not need to examine the code on their own, i.e. they do not need to determine what methods are potentially called within the control flow starting from method `start` in class `ComplexComp`.

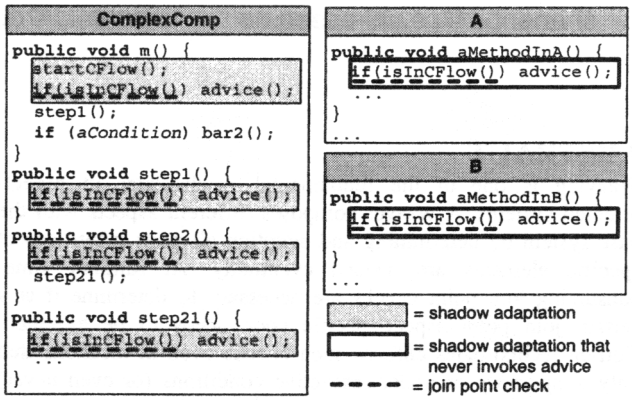

Figure 2. An illustration of the woven tracing aspect including additional classes `A` and `B`.

However, this implementation has some drawbacks due to complete weaving. In general, the exact computation of methods that are executed within a certain control flow is impossible. For example, it is hard to compute upfront whether the condition in method `start` will ever be satisfied and methods `step2`, and `step21` (and methods invoked by `step21`) will ever be invoked from the control flow passing `start`. To guarantee the correct behavior of the aspect the weaver must consider these methods in addition to methods `start` and `step1`, which will be definitively invoked in the control flow. For all these methods the weaver has to determine whether they can also be executed in control flows that do not pass method `start`. In such cases the weaver needs to decorate shadows with join point checks that check at runtime if the current method is part of the control flow to be traced or not. If a large number of different control flows in the application use methods of `ComplexComp` (other than `start`) the join point checks fail most of the time and only cause runtime overhead. If the condition in method `start` is *never* satisfied, the join point checks at method `step2` and `step21` only cause runtime overhead when they are invoked from different methods without ever executing the advice in `TraceComputation` at all. The problem becomes even bigger if `bar21` executes a large number of other methods. The corresponding shadows would also never invoke the advice in `TraceComputation`.

[2] Please note, that the intention here is neither to discuss AspectJ in detail nor to compare the here proposed approach with AspectJ. The intention here is to discuss the impact of complete weaving on the number of join point checks in the woven application.

[3] This use of the `cflow` construct for implementing tracing corresponds (with minor changes) to the implementation like for example proposed in [31]. A similar use for a different purpose can be found for example in [6].

In AspectJ the situation is somewhat different. AspectJ hardly analyzes control flows. For the example from Figure 1 AspectJ determines all methods matching the last part of the pointcut, i.e. (`execution(* *.*(..))`), and creates corresponding join point checks. Since this matches every existing method, AspectJ creates checks for every existing method in the entire system (except for those in system libraries). Figure 2 illustrates the code woven by AspectJ, not only for class `ComplexComputation` but also for two additional classes `A` and `B` also present at weaving-time. Before executing the original code of any method in the system, join point checks are performed. The benefit of this approach is that no cost-intensive computation is necessary which would slow down the compilation process. On the other hand the performance of the whole system decreases in the presence of the woven aspect. This implies that the performance decreases even in classes like `A` and `B` whose methods will never be executed in the control flow of interest. Performance measurements in [4] showed that a single woven `cflow` substantially decreases the overall performance of the system.

The overall problem in the tracing examples is that it is usually not fully computable at weave-time what methods are invoked within the control flow of interest. Consequently, a large number of conditional shadows exist in the system whose execution causes runtime overhead but rarely lead to an execution of the aspect specific code.

2.2 Subject-Observer Protocol

Perhaps the most frequently used example in the area of aspect-oriented programming is the implementation of the *observer design pattern* [9] as discussed for example in [11, 30, 32]. The pattern permits a set of objects (called *observers*) to be attached to other objects (called *subjects*) to become informed about their state changes.

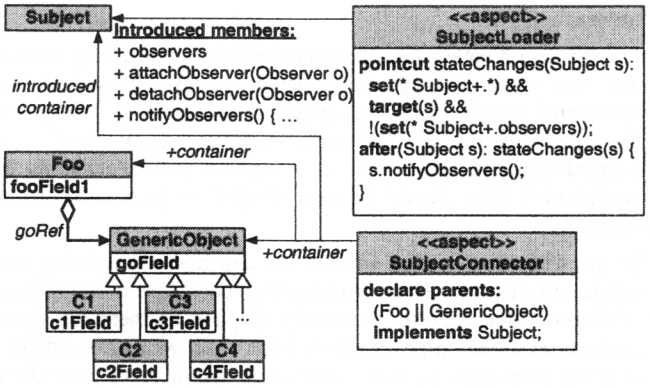

Figure 3. A subject-observer implementation in AspectJ

Observers are interested in state changes, i.e. changes of fields associated with a subject. This includes fields that are directly part of the subject as well as fields of objects which are directly or indirectly referenced by the subject (see for example [14, 11]).

Figure 3 illustrates a typical implementation of the subject-observer protocol in AspectJ based on the *container introduction idiom* [13]. `SubjectLoader` states that observers can be attached to and detached from instances of `Subject` by introducing appropriate fields and methods to `Subject`. The aspect's pointcut `stateChanges` and the corresponding advice define that an assignment to any field declared in `Subject` (i.e. an assignment to a field declared in a class implementing `Subject`) yield the notification of observers. The pointcut language of AspectJ does not permit to declare the state change of every referenced object for a given subject (cf. [14, 11] for further discussion). So, developers have to enumerate explicitly every class whose objects should inform observers about state changes (in `SubjectConnector`). In order to permit the observation of `Foo` instances and its referenced objects of type `GenericObject`, the developer connects `Subject` to both classes (Figure 3).

Again, the implementation suffers from some drawbacks resulting from complete weaving. In general, it is not possible to fully determine what instances are ever referenced by subjects. For example, it is usually not computable if instances of class `C1`, `C2`, etc. are ever referenced by an instance of `Foo` at runtime. Consequently, join point checks need to be created that check at runtime at every field assignment, whether the current object is referenced by an instance of `Foo`. These checks become problematic if a class is frequently used in the application, whose instances are in fact never referenced by a `Foo` at runtime.

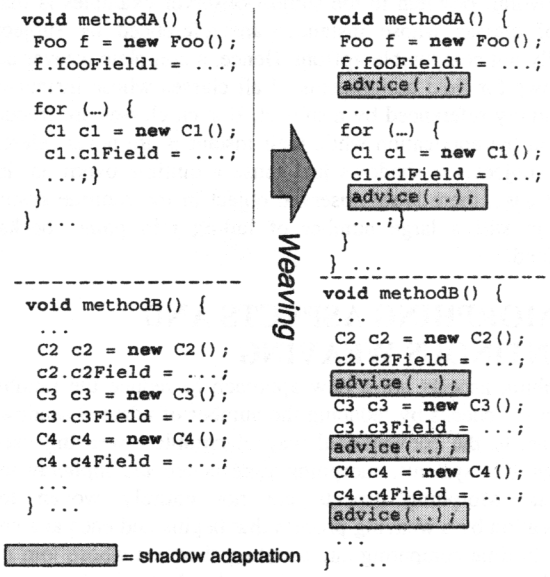

Figure 4. Application using observed classes.

In AspectJ the problem is slightly different. Since AspectJ's pointcut language does not permit to specify classes whose objects are referenced by `Foo`, developers need to add the subject functionality to each class manually (compare to Figure 3). As a result, advice activations are inserted for each state change of instances of `GenericObject` as well as for its subclasses[4]. Figure 4 illustrates an application with a woven subject-observer aspect. Advice activations are created for each assignment of fields declared in `GenericObject` and its subclasses; even in those cases where the instances are not referenced by an instance

[4] Due to limitations of its pointcut language, AspectJ's shadows are unconditional. However, the shadows have to be conditional logically because it must be checked whether an object is referenced by a subject or not.

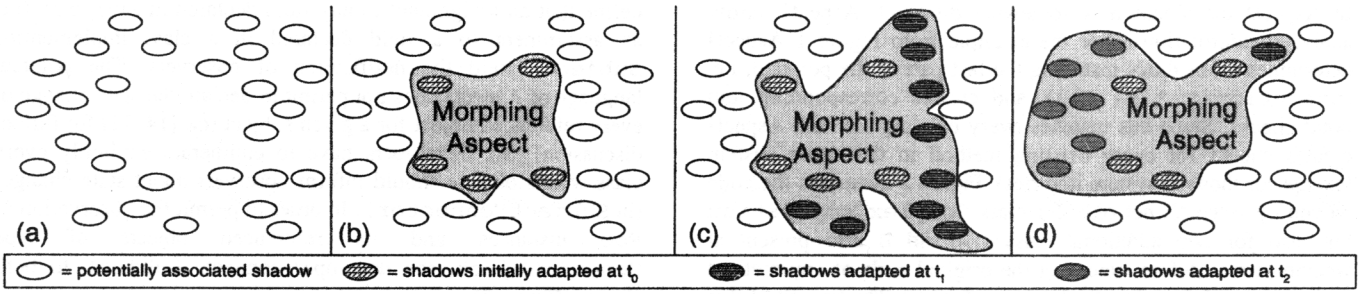

Figure 5. A morphing aspect that changes its set of associated join point shadows during runtime.

of `Foo`. None of the objects one the right hand side are referenced by a `Foo` instance since the objects are newly created. Hence, advice execution is futile. If subclasses of `GenericObject` are frequently used in an application the performance decreases perceivably as every single assignment leads to the execution of the corresponding advice. Typical examples of such often used classes are collection classes or classes that serve as root classes in large frameworks.

The overall problem in the subject-observer examples is that the set of classes whose instances are referenced by subjects is usually not computable upfront. Hence, a complete weaver adapts shadows for field assignments of all classes whose instances are potentially referenced by a subject. If such classes are frequently used in the application while their instances are never referenced by a subject, the shadows just cause a runtime overhead. In the worst case, there is no observed object in the runtime system at all, yet still a large number of failing join point checks are executed.

3. MORPHING ASPECTS AND CONTINOUS WEAVING

Morphing aspects are a new approach to reduce the number of join point checks by reducing the number of adapted shadows. In contrast to the conventional way of complete weaving used by known AO systems, morphing aspects are *incompletely woven* aspects. Morphing aspects are not entirely woven to an application by a weaving process that begins and ends at a certain point in time computing and adapting shadows whose join points possibly execute aspect-specific code. Instead, the necessary shadows to be adapted are continuously computed and adapted (or released) by the aspects itself at well-defined points in the execution of the program, i.e. at certain join points. When a morphing aspect is woven it starts with a small set of initial join point shadows and dynamically adapts or releases shadows just when they are needed. Hence, the number of shadows associated with a morphing aspect changes during the aspect's lifetime. We call this process of computation, adaptation and release of an aspect's shadows *morphing*. We refer to the whole weaving process, i.e. initial weaving of morphing aspects, the morphing during their lifetime and unweaving as *continuous weaving*.

Figure 5 illustrates a morphing aspect and its set of join point shadows at runtime. The ovals represent all join point shadows which are potentially associated with an aspect during its lifetime as they would have been computed during complete weaving. The ovals within the aspect's border represent shadows adapted for the aspect. As long as the aspect is not woven, there are no shadows adapted for the aspect (Figure 5a). Initially, when the developer weaves the morphing aspect, a relatively small number of join point shadows is adapted by the aspects (Figure 5b). The set of actually adapted shadows changes during the aspect's lifetime. At a later point in time (Figure 5c) the aspect has nine more shadows in addition to the original join points. Even later (Figure 5d) five more shadows were adapted and most of the previous ones were released. In contrast to this, a completely woven aspect adapts all shadows which are potentially associated with an aspect (and creates corresponding join point checks), i.e. all ovals in Figure 5 right from the beginning. Morphing aspects adapt fewer shadows in the system. Hence, morphing aspect cause less runtime overhead due to failing join point checks as there are fewer join point checks in the system.

As a key characteristic of morphing aspects they themselves determine at runtime at what points in the execution of the program the adaptation or release of join point shadows is necessary. Hence, join points in morphing aspects serve two different purposes. On the one hand the aspect's functionality (like logging, or notification of observers) is invoked, on the other hand the morphing process is started whenever particular join points are reached.

For the specification (and implementation) of morphing aspects and the corresponding morphing processes, developers are confronted with the following questions: What are the join points the aspect gets initially woven to, i.e. what shadows need to be adapted initially? When does the morphing process needs to be carried out? How should the new set of join point shadows be determined?

In the following section we discuss dependencies among join points and shadows. Those dependencies determine a minimal set of shadows that need to be initially adapted. Furthermore, these dependencies determine what join point shadows can be adapted at some later point in time. Afterwards, we describe how these properties can be utilized to specify the morphing process.

3.1 Join Point Dependencies

In order to determine when new shadows need to be adapted or can be released, developers of a morphing aspect have to analyze how those join points (and their shadows) which are relevant for the aspect to be specified depend on each other. Dependencies among join points describe that a certain join point associated to an aspect (the *dependent* join point) can only be reached if another join point associated with the same aspect has been reached before. We call the corresponding shadows *dependent shadows*. All join points that do not depend on any other join

point are *independent* (and are represented by independent shadows). On the technical level a dependency between join points expresses that join point checks of dependent shadows fail as long as the join points they depend on have not been reached before. Consequently, shadows for dependent join points do not need to be adapted as long as the join points they depend on are not yet reached. This allows the adaptation of dependent shadows to be shifted to a later point in time. This situation is different for independent shadows. Their join points potentially occur in the execution of a program independently of any other join point associated to the same aspect and the adaptation of their shadows cannot be shifted to a later point in time. Hence, when the morphing aspect is initially woven at least all independent join point shadows need to be adapted.

In the following we illustrate dependencies among join point shadows for the examples presented in section 2.1 and 2.2. We represent a potential join point shadow by an oval whereby an oval's label describes the shadow. A directed edge from a shadow A to a shadow B represents a dependency relationship which expresses that the join point represented by shadow A depends on the join point represented by shadow B. This also implies that shadow A depends on shadow B. The edge's label describes the kind of dependency. A shadow without any outgoing edge is an independent shadow, while a shadow with at least one outgoing edge is a dependent shadow.

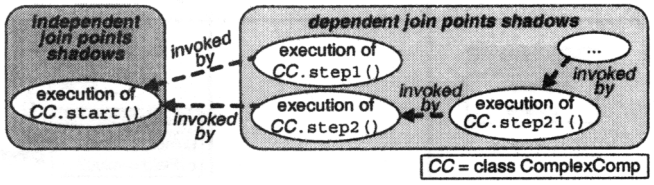

Figure 6. Dependencies in the tracing example

Figure 6 illustrates the dependencies among join points shadows for the tracing example introduced in section 2.1. The execution of method `step1` or the execution of `step2` only need to lead to an execution of aspect-specific code if method `start` in class `ComplexComp` is executed and `start` invokes either `step1` or `step2`: the join point checks for the shadows at `step1` and `step2` fail as long as method `start` is not executed and invokes `step1` or `step2`. Hence, both shadows directly depend on the shadow for method `start`. For the same reasons the shadow at `step21` depends directly on the shadow at `step2` and the shadows for all methods that are eventually invoked by `step21` depend the shadow at `step21`. In the tracing example, all join point shadows either directly or indirectly depend on the shadow representing the join point for the execution of method `start`. The shadow at `start` does not depend on any other shadow, i.e. this is an *independent shadow*. On a more abstract level, shadows of all methods that are either directly or indirectly invoked by `start` depend on the shadow for `start`.

The dependencies of join points in the subject-observer aspect are slightly more complex (Figure 7). The join points (and their shadows) to be handled by the aspect are the state changes of subjects (instances of `Foo`) and their referenced objects. So, all assignments to fields declared in `Foo`, `GenericObject` and subclasses of `GenericObject` are join point shadows, which are potentially associated with the aspect. Those assignments execute aspect-specific code only if there is at least one object observing a `Foo` instance. This in turn depends on invocations of method `attach` which registers observers. Hence, all shadows for `fooField` and `goRef` assignments depend on the shadow at method `attach`[5]. The same is true for assignments to fields declared in `GenericObject` and its subclasses. However, their dependency is more complex. First, when an observer is attached, those shadows need to be adapted only for classes whose instances are referenced by a `Foo`. For example as long as no `GenericObject` instance is referenced by a `Foo` `goField` assignments need to be adapted. Second, assignments to `goField` depend on the `goRef` assignment since an instance of `GenericObject` becomes referenced by an instance of `Foo` by assigning it to `goRef`. For the same reason all further assignments to fields declared in subclasses of `GenericObject` depend on the `goRef` assignment.

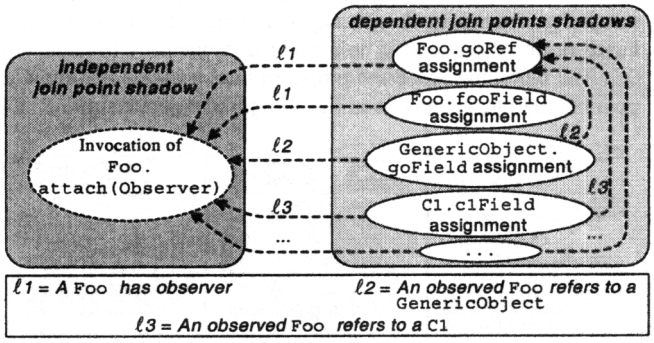

Figure 7. Dependencies in the subject-observer example

As exemplified in Figure 7 the only independent join point is the one for the invocation of method `attach`. In the subject-observer example the different natures of join points associated to the subject-observer aspect becomes manifest: the field assignments depend on a join point which does not lead to an execution of the aspect-specific code, because field assignments depend on the execution of method `attach`. A field assignment join point informs the observers about a state change, while the join point at method `attach` does not. In order to emphasize that fact, we rendered the `attach` shadow using a different style (Figure 7).

3.2 Specifying the Morphing Process

Once the dependencies are determined developers have to decide how to utilize them for the specification of a morphing aspect's morphing process. First, developers have to specify what shadows including corresponding join point checks are to be initially handled. Next, developers have to specify what initial join points start the morphing process. Then, developers have to define the morphing process itself.

At least all independent join point shadows have to be initially adapted, because they do not depend on any other shadows.

[5] For the same reason, field assignments depend on method `detach`, because if after invoking `detach` no object observes `Foo` no aspect-specific code need to be executed. For reasons of simplicity we omitted this dependency in Figure 7.

Hence, it is not possible to utilize their dependencies for a late shadow adaptation. Additionally, the developer can decide to adapt some additional dependent shadows at initial weave time: In case the dependent join points are reached very often, the developer may not want to adapt their shadows just during the weaving process but right from the beginning.

The morphing process consists of the following parts. First, the process has to determine a number of dependent shadows to be adapted (or to be released). For that purpose, the morphing process can make use of *reflection* [21]: the process reflects on the join point starting the morphing process and computes the dependent shadows. Second, the morphing process specifies the join point checks for all shadows to be created. Third, it has to be determined for each newly adapted shadow whether it's join points invoke aspect specific code and/or the morphing process.

In the following we illustrate two reasonable specifications of the morphing process for the tracing and subject-observer examples based on the discussion of join point dependencies from section 3.1. The morphing processes proposed here are kept as simple as possible to illustrate how to utilize join point dependencies.

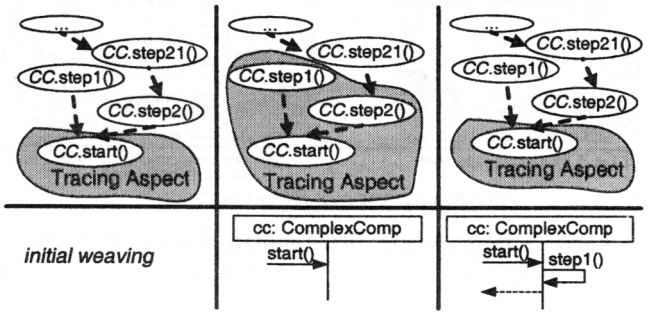

Figure 8. Morphing tracing aspect after initial weaving, after execution of `start`, and after execution of `step1`.

For the tracing aspect (Figure 8) at least one (independent and unconditional) shadow at method `start` in `ComplexComp` has to be initially adapted. For reasons of simplicity, we decided to adapt only this shadow to keep the number of join point checks low and to adapt all dependent shadows as late as possible. Hence, Figure 8 illustrates that only one shadow is initially adapted for the tracing aspect.

Once a join point of this shadow is reached, the tracing code is executed and the morphing process starts. The morphing process determines all methods that potentially are invoked by the method enabling the process. The shadows for all these methods are adapted. The corresponding join point checks examine if the method is invoked within the control flow being traced[6]. If the join point check succeeds, the tracing code is executed and the morphing process starts once again. Whenever a method within the control flow is left, all dependent shadows are released. So, if a `ComplexComputation` receives a message `start` (and the message is logged), the morphing process computes all methods that are potentially invoked by `start` and adapts the corresponding shadows (`step1` and `step2`) (in the middle of Figure 8). When `step1` is invoked by start the method is logged and the morphing process starts once again. Since no other methods are potentially invoked by method `step1` no further shadows are created. If `step2` is not invoked and `start` is left the shadows at `step1` and `step2` are released (right hand side of Figure 8). So, as long as the condition in method `start` does not lead to an execution of `step2`, no shadows for `step21` (and methods invoked by `step21`) are created.

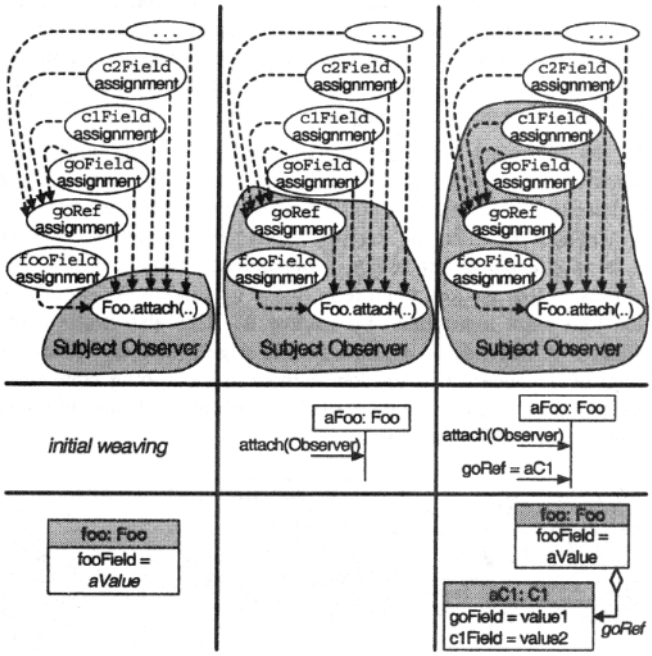

Figure 9. Subject-observer as a morphing aspect at initial weave time, after observer attachment, and after assigning an instance of `C1`.

For the subject-observer aspect at least one unconditional shadow for method `attach` needs to be initially adapted[7] (see left hand side of Figure 9). Similar to the previous example we simply decided to adapt only this shadow to keep the number of adapted join point shadows low (and to simplify the morphing process). An invocation of the method `attach` in class `Foo` starts the morphing process.

A simple morphing process for this aspect works as follows. First, the process adapts shadows for all assignments to fields `fooField` and `goRef` whose execution leads to notifications of observers. Second, the process reflects on the `Foo` instance whose join point started the morphing process. It determines the referenced object and adapts the field assignment shadows for notifying observers. And finally, the morphing aspect adapts the `goRef` join point to start the morphing process.

Figure 9 illustrates the above described morphing process. After an observer is attached the morphing process adapts shadows for all assignments to fields declared in `Foo`. Furthermore, the

[6] There are different ways to implement such a condition. In AspectJ the current thread is stored when the control flow starts, and each join point check determines whether the current thread is stored. Languages like for example Smalltalk permit to analyze the call stack to determine whether the current method occurs in the control flow of interest.

[7] Like in the previous section we skip for reasons of simplicity the discussion about method `detach` here which is also an independent shadow.

morphing process determines the object referenced by `goField` of object `foo`. Since in Figure 9 `foo` does not refer to any `GenericObject` no further shadows are adapted. When an instance of `C1` is assigned to `foo` the morphing process starts once more (because assignments of `goRef` start the morphing process). The process determines the fields of the assigned object. Since `c1` is an instance of `C1`, shadows are adapted for all assignments to `goField` (declared in `GenericObject`) and `c1Field` (declared in `C1`).

4. IMPLEMENTATION EXAMPLE

In this section we introduce an exemplary implementation of a morphing aspect in AspectS (see [16] for a detailed introduction in AspectS). AspectS is an aspect-oriented system providing dynamic weaving in the Smalltalk dialect Squeak. We concentrate here on the implementation of the tracing aspect. For the morphing implementation of the subject-observer implementation, please refer to [14] and [3].

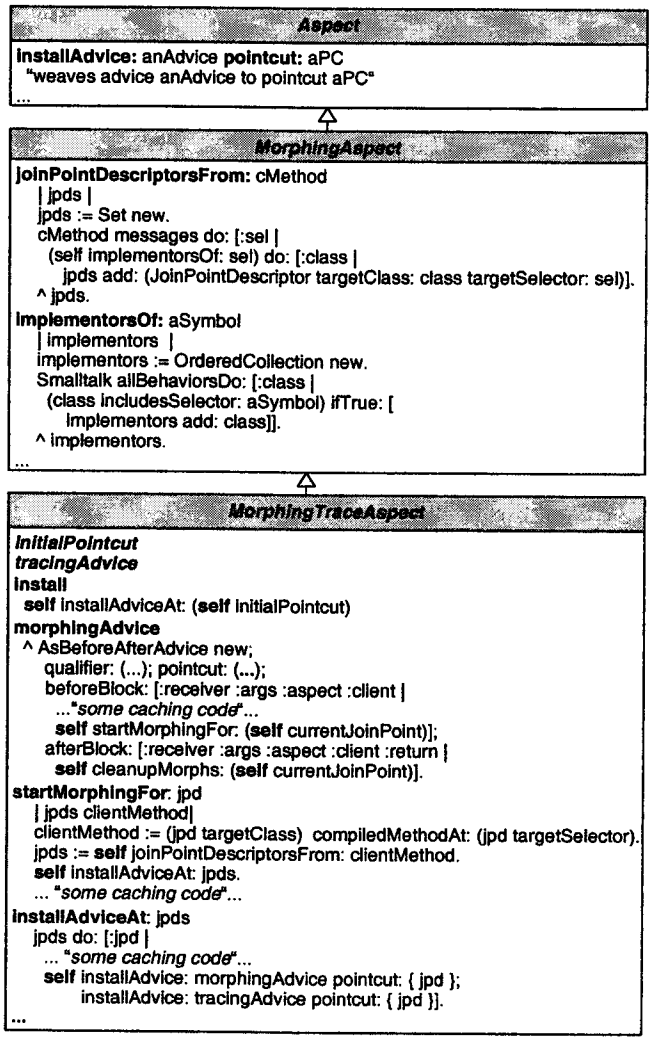

Figure 10. Tracing as a abstract morphing aspect in AspectS

AspectS is based on *method wrappers* [5]. A shadow for a method execution join point or a method call join point is adapted by wrapping the receiving method. The method to be wrapped is specified by a *join point descriptor* (instance of `JoinPoint Descriptor`) which refers to a class and to a method selector. Advice directives in AspectS are runtime objects that refer to a pointcut. Pointcuts are collections of join point descriptors. If advice directives are installed at runtime, all methods referenced by the join point descriptors are wrapped. The wrappers handle execution of qualifiers (which correspond to join point checks) and the execution of the advice. Advice is implemented by blocks (see [10] for an introduction to Smalltalk blocks, see [16] for a detailed description of how advice objects are created and executed using blocks).

Tracing aspects based on morphing aspects are subclasses of the (abstract) class `MorphingTraceAspect` (Figure 10). `MorphingTraceAspect` contains the abstract method `initialPointcut` that returns the set of join point descriptors specifying the independent join points whose shadows need to be initially adapted.

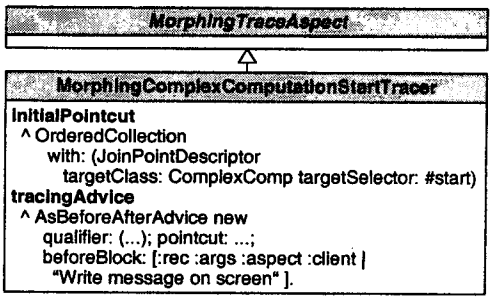

Figure 11. Concrete tracing aspect as a morphing aspect.

The aspect refers to two advice objects, both returned by corresponding methods `tracingAdvice` and `morphingAdvice`. The (abstract) method `tracingAdvice` returns the advice to be executed during tracing, `morphingAdvice` provides the advice starting the morphing process. Our morphing advice contains two blocks which are invoked before and after the corresponding join point is reached. The before block starts the morphing process on the given join point by invoking method `startMorphingFor`. Method `startMorphingFor` determines the runtime-object for the invoked method and computes all join points descriptors that depend on that method (see methods `joinPointDescriptorsFrom:` and `implementorsOf:` in class `MorphingAspect` in Figure 10). A shadow is adapted for each of those join point descriptors that invokes the morphing advice as well as the tracing advice (method `installAdviceAt:pointcut:`). In order to use the morphing trace aspect developers have to extend `MorphingTraceAspect` and override `initialPointcut` and `tracingAdvice`. Figure 11 illustrates a sample class `MorphingComplexComputationStartTracer`. A tracing aspect is initially woven by instantiating the corresponding class and invoking method `install`.

Creating and integrating shadows (i.e. method wrappers) in AspectS is a time-consuming task, yet to be optimized. Hence, it

[8] According to the Smalltalk meta-object protocol (cf. [10]) method implementations have runtime representations.

is typically not desirable to start the morphing process at every possible join point. For example, if the methods to be traced are executed quite often, the time consumed for the morphing process can be higher than the benefit of dismissed join point checks. Therefore, the implementation of the tracing aspect in AspectS uses *lazy morphing* by default. Dependent shadows are adapted whenever the join points they depend on are reached and the morphing process did not already start at these join points. Lazy morphing does not release shadows during morphing. Instead, shadow adaptations reside in the system until the developer uninstalls the whole aspect. So, every execution of the morphing process potentially increases the number of adapted shadows for the aspect, but does not delete any.

The use of lazy morphing turned out to be practical in a number of experiments. Those experiments showed that the number of shadows adapted by lazy morphing is still significantly smaller than a complexly woven trace aspect.

5. EXPERIMENTAL RESULTS

Table 1 summarizes a number of performance measurements in AspectS on a Pentium 4.2 GHz with the Squeak Virtual Machine version 3.4.4. The Smalltalk image contained the Comanche Http Server [7] as well as the Squeak CommandShell [8]. Overall, the image contained more than 2200 classes with more than 35000 compiled methods. We measured 100000 times the execution time for the adaptation and release of shadows, the execution time for (empty) methods and the execution time for methods with adapted shadows whose join point checks always pass as well as with shadows whose checks always fail. We implemented the adapted shadow with the successful join point check by an empty around advice whose join point condition immediately succeeds without any additional computation. We implemented the adapted shadow with the failing join point check by a join point condition that immediately fails without any additional computation. Our measurement showed that the execution of an unconditional shadow was approximately 90 times slower than the execution of an empty method. The execution of a dead shadow was about 9 times slower than the execution of an unconditional shadow[9].

Table 1. Experimental Results for shadow creation, deletion and method execution time (in ms) in AspectS

	Average	Minimum	Maximum
Single shadow adaptation	0.1813	0.1756	0.1998
Single shadow adaptation	0.3113	0.1259	0.3445
Method execution (ME) without adapt.	0.0004	0.0002	0.0005
ME with successful join point check	0.0368	0.0346	0.0371
ME with failing join point check	0.3269	0.2555	0.3282

Next, we created a single (lazy) morphing tracing aspect to trace the execution commands in the command shell. The corresponding advice simply wrote all messages to the screen. The initial weaving of the tracing aspect adapts just a single join point shadow and took 0.45 milliseconds. As soon as the method to be traced has been invoked for the first time, the morphing process started for 35 times creating 253 shadows. This process took about 9.5 seconds. Starting the control flow afterwards did not lead to any additional execution of the morphing process. From then on the execution of the method to be traced took about 2.5 seconds.

We compared this result with a corresponding complete weaving (see Table 2). The computation of all potentially invoked methods was not practicable (the computation took more than 3900 seconds). Hence, we did the same approach like AspectJ to weave the aspect to all existing methods in the image (except some system methods). To do so, we wove the aspect to more than 35000 methods. This complete weaving took about 6.8 seconds. The control flow execution afterwards took the same time like the morphing tracing aspect.

Table 2. Tracing in an experimental environment as morphing aspect and completely woven aspect

	# Adapt.	Time
Morphing Tracing Aspect, Initial Weaving	1	0.45 ms
Complete Weaving with Shadow computation	7930	3949 s
Complete Weaving (no Shadow computation)	35852	6.82 s
1st cflow execution in Morphing Aspect	253	9.54 s
1st cflow execution in Non-Morphing Aspect	35852	2.51 s
cflow execution in Morphing Aspect	253	2.51 s
cflow execution in Non-Morphing Aspect	35852	2.51 s

Table 3. Consumed Time for tracing control flow in a completely, and an incompletely woven aspect.

	Time
Complete Weaving and first cflow execution:	9.33 s (=6.82 s + 2.51 s)
Incomplete Weaving and first cflow execution:	9.54 s (=9.54 s + 0.45 ms)

As a result, this experiment showed that the initial weaving of the morphing aspect and the first tracing of the control flow took about 9.54 seconds while the completely woven aspect and a first execution of the control flow took about 9.33 seconds (Table 3). The difference of 0.19 seconds is the price for using a morphing aspect instead of a completely woven one. However, the number of adapted shadows by using a morphing aspect is only 1 % of the number of adapted shadows of the completely woven aspect. These shadows decrease the performance of the whole system, because each shadow whose join point check fails cause the runtime overhead of more than 0.3 milliseconds (according to Table 1). Preliminary experiments showed for example that the response time of the http server contained in the image was a few hundred times slower than before weaving the tracing aspect, because weaving the tracing aspect according to the weaving strategy of AspectJ also adapted a large number of shadows even in those classes that will never be invoked in the control flow to be traced.

6. RELATED WORK

Dynamic weaving in combination with *just-in-time aspects* as proposed in [26, 27] is closely related to morphing aspects. Just-in-time aspects are dynamically woven to the system when they are really needed. Furthermore, just in time aspects are woven to the application in *one atomic step* (see [27], page 101). Consequently, just in time aspects do not perform any additional

[9] The reason for the slow execution of dead shadows lies in the way how wrappers and wrapped methods are implemented. Wrappers store the wrapped method in a field. When a join point check fails the original method is executed by calling the time-consuming *value:* method.

join point checks as long as they are not woven. In that way just in time aspects overcome the problem of unnecessary shadows in comparison to static weaving. Nevertheless, just-in-time aspects are woven completely because of the atomicity property. Hence, after dynamically (and completely) weaving an aspects the problem of unnecessary shadows arises just like in static woven systems.

Another approach that relates to our work on morphing aspects is the *selective just-in-time weaver* as proposed in [28], an extension to the work of just-in-time aspects. The (Java based) selective weaver permits developers to choose between two different kinds of join point shadows: either as breakpoints in the JVM or as statically embedded hooks. While breakpoints can be created much faster, their execution is time consuming (see [28] for a detailed discussion on the performance issues). Embedded hooks on the other hand execute faster while their creation is quite slow in comparison to that of breakpoints. Selective just-in-time weavers try to overcome the performance overhead caused by frequently executed shadows by embedding such shadows statically. From that point of view a selective weaver and morphing aspects are similar. The selective weaver causes a performance overhead for embedding hooks in order to achieve a performance advantage for the further execution of the program. Similarly, the morphing process executed by morphing aspects causes a performance overhead to achieve a performance advantage for the further execution of the program. However, the main difference between both approaches is that a selective weaver does not reduce the number of conditional shadows.

The virtual machine *Steamloom* [4] belonging to the aspect-oriented language *Caesar* [23] also tackles the problem of time-consuming join point checks. Steamloom implements join point checks and advice invocations at the virtual machine level. In [4] the performance of advice making use of the join point checks on VM level and the statically woven aspects in AspectJ based on the cflow construct is measured. The result shows that the Steamloom VM has a significant performance advantage over the completely woven approach of AspectJ. The intention of Steamloom and morphing aspects is very similar, since both tackle the performance overhead caused by join point checks. The difference between Steamloom and the implementation of morphing aspects as proposed in this paper is that while weaving in Steamloom is performed by redirecting messages at the VM level our AspectS-based implementation carries out changes to the runtime representation of methods at the application level.

Besides the approaches that provide pure dynamic weaving there are also approaches that remove unnecessary runtime checks based on a static analysis. For example [22] describes a *partial evaluator* based on the definitional interpreter specified in [33] to reduce the number of unnecessary join point checks. In [29] a reduction of join point checks is achieved by a static analysis of the call stack. Currently, we do not have any experimental results that compare the number of failing join point checks caused by these approaches with the number of failing join point checks caused by morphing aspects within an experimental environment.

7. DISCUSSION AND CONCLUSION

In this paper we addressed the problem of unnecessary join point shadows caused by complete weaving. We motivated the problem by illustrating two typical examples for aspect-oriented programming and their implementation in the aspect language AspectJ.

We proposed morphing aspects to overcome the problem of unnecessary join point checks. Morphing aspects are incompletely woven aspects that change their set of join point shadows at runtime based on a continuous weaving process. With incomplete weaving, not every shadow within the base system whose join points potentially execute aspect-specific code is adapted. Instead, morphing aspects utilize dependencies among join points and their shadows that permit to delay the adaptation of shadows just to the point when join points they depend on are reached. As a result, the number of adapted shadows of a morphing aspect is much smaller in comparison to that of completely woven aspects. This is because dependent join point shadows are not adapted initially, but at a later point in time when they are actually needed. Experiments with morphing aspects in the aspect-oriented system AspectS showed that by using morphing aspect the number of join point shadows is significantly reduced. In that way the performance overhead caused by failing join point checks is reduced, too. However, it should be noted that the performance overhead of join point checks in AspectS is quite high as shown in section 5. Hence, the benefit of morphing aspects in such a system is much higher than in systems where join point checks are less expensive.

The benefit of realizing an aspect as a morphing aspect depends on a number of influencing factors. In general, a prerequisite for the successful application of morphing aspects is a large number of failing join point checks during the execution of a program. According to the example we gave in section 2, such prerequisite is fulfilled if, for example, a tracing aspect is to be implemented in an application with a large number of threads that never invoke the method where tracing should begin. Also, such prerequisite is fulfilled if instances of a class are only very rarely observed during the execution of a program. The prerequisite is usually not fulfilled, if the aspects in the system hardly rely on join point checks, i.e. if the woven application mainly consists of unconditional join point shadows.

The morphing process needs additional time to determine and create dependent join point shadows. Developers must trade-off between the runtime overhead caused by unnecessarily introduced runtime checks caused by unnecessary adapted shadows and the overhead caused by the morphing process itself.

Morphing aspects impose a number of requirements on the underlying aspect-oriented system. This restricts their application to a number of systems. As most fundamental requirement the underlying system must permit dynamic weaving, i.e. weaving of aspects during runtime. A number of systems such as *PROSE* [26, 27], *AspectS* [16], *JAC* [25], *Object Teams* [32, 15], or Caesar [23] fulfill this requirement while systems like *AspectJ* [18] or *Sally* [12] do not. As another requirement morphing aspects typically require the computation of dependent shadows at runtime, i.e. the shadows to be associated with an aspect are statically not known. However, not every system providing dynamic weaving permits the computation of join points at runtime. For example, Object Teams assumes that the shadows are statically declared.

As far as we know there is currently no approach like morphing aspects and continuous weaving that utilizes dependencies among

join points and join point shadows to determine join point shadows to be adapted (or released) during an aspect's lifetime.

As future work morphing aspects and selective weavers [28] should be combined in order to gain the benefit of both reducing unnecessary shadows as well as reducing the execution time of join point shadows. Thereto, it needs to be analyzed how far the morphing idea can applied to more static and complex languages like Java which provide only limited reflective capabilities.

8. REFERENCES

[1] Aksit, M. (ed.): *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, MA, March 17 - 21, ACM, 2003.

[2] Aksit, M.; Mezini, M.; Unland, R. (eds.): *Objects, Components, Architectures, Services, and Applications for a Networked World*, LNCS 2591, Springer-Verlag, 2003.

[3] AspectS, version 0.5, Homepage, http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/

[4] Bockisch, C.; Haupt, M.; Mezini, M.; Ostermann, K.: *Virtual Machine Support for Dynamic Join Points*, 3rd International Conference on Aspect-Oriented Software Development (AOSD), Lancaster, UK, March, 2004.

[5] Brant, J., Foote, B., Johnson, R. E., Roberts, D.; *Wrappers to the Rescue*, In: Proceedings of the 12th European Conference on Object-Oriented Programming ECOOP, LNCS 1445, Springer-Verlag, 1998, pp. 396-417.

[6] Cibran, M.; D'Hondt, M.; Jonckers, V.: *Aspect-Oriented Programming for Connecting Business Rules*. In: Proc. of the 6th International Conference on Business Information Systems (BIS'03). Colorado Springs, USA, June 2003.

[7] Comanche http server, version 6.1, http://squeaklab.org/comanche/httpserver/index.html

[8] CommandShell for Squeak - Version 3.0.1, http://minnow.cc.gatech.edu/squeak/1914

[9] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[10] Goldberg, A.; Robson, D.: *Smalltalk 80 – The Language and its Implementation*, Addison-Wesley, 1983.

[11] Gybels, K.; Brichau, J.: *Arranging Language Features for More Robust Pattern-based Crosscuts*, In: [1], pp. 60-69.

[12] Hanenberg, S.; Unland, R.: *Parametric Introductions*, In: [1], pp. 80-89.

[13] Hanenberg, S.; Schmidmeier, A.; Unland, R.: *AspectJ Idioms for Aspect-Oriented Software Construction*, 8th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany, June 25–29, 2003.

[14] Hanenberg, S.; Hirschfeld, R; Unland, R.: *Aspect Weaving: Using the Base Language's Introspective Facilities to Determine Join Points*, In: Workshop on Advancing the State-of-the-Art in Run-Time Inspection (at ECOOP), 2003, http://www.st.informatik.tu-darmstadt.de/pages/workshops/ASARTI03/ HanenbergASARTI03.pdf.

[15] Herrmann, S.: *Object Teams: Improving Modularity for Crosscutting Collaborations*, In: [2], pp. 248-264.

[16] Hirschfeld, R.: *AspectS – Aspect-Oriented Programming with Squeak*, In: [2], pp. 216-232.

[17] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwing, J.: *Aspect-Oriented Programming*. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, 1997, pp. 220-242.

[18] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold William G.: *An Overview of AspectJ*, In: Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 2072, Springer-Verlag, 2001, pp. 327-353.

[19] Kiczales, G. (ed.): *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 22-26, ACM, 2002.

[20] Lopes, C: *AOP: A Historical Perspective*. In: Filman, R.; Elrad, T.; Aksit, M.; Clarke, S. (eds.): Aspect-Oriented Software Development, Addison-Wesley, 2004 (to appear).

[21] Maes, P.: *Concepts and Experiments in Computational Reflection*, Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Orlando, Florida, 1987, pp. 147 - 155.

[22] Masuhara, H.; Kiczales, G.; Dutchyn, C.: *A Compilation and Optimization Model for Aspect-Oriented Programs*, Proceedings of Compiler Construction (CC2003), LNCS 2622, Springer-Verlag, 2003, pp.46-60.

[23] Mezini, M.; Ostermann, K.: *Conquering Aspects with Caesar*, In: [1], pp. 90-99.

[24] Ossher, H.; Tarr, P.: *Using multidimensional separation of concerns to (re)shape evolving software*. Communication of the ACM, 44 (10), 2001, pp. 43-50.

[25] Pawlack, R.; Seinturier, L.; Duchien, L. Florin, G.: *JAC: A Flexible Solution for Aspect-Oriented Programming in Java*, Proceedings of Reflection 2001, Kyoto, Japan, September 25-28, 2001, LNCS 2192, Springer, 2001, pp. 1-24.

[26] Popovici, A.; Gross, T.; Alonso, G.: *Dynamic Weaving for Aspect-Oriented Programming*, In: [19], pp. 141 – 147.

[27] Popovici, A.; Gross, T.; Alonso, G.: *Just in Time Aspects*, In: [1], pp.100-109.

[28] Sato, Y; Chiba, S.; Tatsubori, M.: *A Selective Just-in-Time Aspect Weaver*, Proceeding of the Second International Conference on Generative Programming and Component Engineering (GPCE), Erfurt, Germany, September 2003, pp. 189-208.

[29] Sereni, D; de Moore, O.: *Static Analysis of Aspects*, In: [1], pp. 30-39.

[30] Stein, D.; Hanenberg, S.; Unland, R.: *A UML-based aspect-oriented design notation for AspectJ*, In: [19], pp. 106 - 112.

[31] Skotiniotis, T., Lieberherr, K., Lorenz, D. H.: *Aspect Instances and their Interactions*, Workshop on Software-engineering Properties of Languages for Aspect Technologies at AOSD'03, http://www.daimi.au.dk/~eernst/splat03/, 2003

[32] Veit, M.; Herrmann, S.: *Model-View-Controller and Object Teams: a Perfect Match of Paradigms*, In: [1], pp. 140-149.

[33] Wand, M.; Kiczales, G.; Dutchyn, C.: *A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming*. to appear in ACM Transactions on Programming Languages and Systens *(TOPLAS)*, 2003.