Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computer Languages, Systems & Structures 32 (2006) 87–108
COMPUTER LANGUAGES, SYSTEMS & STRUCTURES
www.elsevier.com/locate/cl

Open Aspects

Robert Hirschfeld[a,*], Stefan Hanenberg[b]

[a] *DoCoMo Euro-Labs, Future Networking Lab, Landsberger Strasse 312, 80687 Munich, Germany*
[b] *University of Duisburg-Essen, Department of Computer Science, Schützenbahn 70, 45117 Essen, Germany*

Received 13 September 2005

Abstract

Open Aspects are our approach to face unplanned changes in systems that are based on aspect-oriented composition at runtime. They support explicit adaptation models, allowing developers to describe system change events to be observed, and corrective actions to be taken. These events and actions cover both the base system affected by aspects as well as the aspects affecting the base system themselves. The proper combination of change events and corrective actions allows for conditional just-in-time runtime re-composition. This paper offers a detailed discussion of difficulties related to change in aspect-oriented systems and a description of consistency constraints inherent to them. An implementation illustrating Open Aspects and their application is provided.

Keywords: Aspect-oriented programming; Dynamic aspects; Open aspects; Runtime weaving

0. Introduction

Systems utilizing aspect-oriented programming (AOP) differ in how and when they carry out the processes of composing aspects into the base system, a process also known as weaving. There are systems that statically weave at compile or load-time. Other systems permit the composition of aspects at an application's runtime.

We usually prefer to carry out changes to our systems while they are offline so that the detection and resolution of problems that become apparent during aspects composition will not interfere with the running system. However, there are situations where such practice is not desirable, is inadvisable, or even impossible due to domain specific requirements to the system in question. In telecommunications, for instance, system downtime results in disruption of services, leading to less customer satisfaction, and because of that has to be kept to a minimum or to be avoided entirely [1–3]. Ambient and embedded computing infrastructures and environments are yet another example of systems that require online adaptations—changes to the running system. The reason for weaving in aspects dynamically results from the requirement that the system aspects to be woven are expected to change at runtime.

Recent work of the aspect-oriented software development community indicates that dynamic aspects are becoming of increased interest. Dynamic aspects offer compositions that can be made effective or revoked at runtime. Prominent activities in this research area are efforts to provide technologies for dynamic method call interception (MCI [4]) or extensions to virtual machines (VM) for enhanced method call dispatch. Systems like PROSE [5,6], Steamloom [7], JAC

* Corresponding author. Tel.: +49 160 478 5212; fax: +49 89 56824 300.
E-mail address: hirschfeld@acm.org (R. Hirschfeld).

1477-8424/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2005.09.003

[8], AspectL [9], or AspectS [10] are all concerned with hot-deployment of aspects. They employ runtime weaving to dynamically add new code, modify or remove available code or change the way the base application is interpreted. Their weaver considers the systems or code segments to be combined at one particular point in time. Pointcut expressions or predicates (both terms are used interchangeably in this text) are evaluated to compute sets of join-point shadows [11] to be instrumented, and integration steps are performed as necessary to provide the desired composed behavior. Join-point shadows are roughly correspondent locations of actual join-points in a program's representation such as the program's source or its meta structure. We consider poincut predicates to be one essential contribution of AOP to generically designate subsets of a system's computational properties. Examples for that are all accesses to an instance variable, all sends are receptions of a message, or all messages sent by a group of senders or received by a specific receiver.

Opening up systems and allowing them to be changed after their initial deployment—possibly by code providers other than the original one and probably while they are running—increases the likelihood of system changes not planned for at the beginning of their design and development. Furthermore, in such open systems the point in time changes can happen as well as their order and frequency are undetermined. Presence and characteristics of classes, instances, or methods can be revised at any instant or not at all. Furthermore, we can assume changes that not only address elements that belong to the base system, but to affect aspects themselves. Changing pointcuts and their associated sets of join-point shadows or changes to advice code is an example.

Changes like that can and will have an effect on AOP-induced invariants as mentioned above. Hence, a mechanism is needed to explicitly maintain these invariants.

As a simple illustration let us look at a system that uses classes not known at compile time but loads them dynamically on demand. New classes not known during the development and initial deployment of the original system thus appear. In such a situation the problem from the aspect-oriented point of view is that it is not clear how the system should behave. Should pointcut coverage be monitored, and, if necessary, should aspects be recomposed? Or should such changes be ignored at all? Questions like that are not addressed by current approaches and technologies.

Open Aspects is our approach to handling unplanned system changes at runtime. Open Aspects support explicit adaptation models, allowing developers to describe system change events to be observed and corrective actions to be taken in response to these events. System change events and corrective actions cover both the base system affected by aspects and the aspects themselves affecting the base system. The proper combination of change events and corrective actions allows for conditional just-in-time runtime re-composition.

Contributions of our paper include:

- A description of consistency constraints inherent to aspect-oriented systems.
- A detailed discussion of the associated change problem.
- A solution to this change problem by separating and providing an explicit adaptation model to aspect developers.
- An implementation illustrating our solution and its application.

In the next section we give a motivating example. In Section 3, we explain open aspects in general. Section 4 demonstrates how Open Aspects work in the presence of change. Section 5 describes openAspectS, our implementation of Open Aspects in AspectS. Section 6 shows an application example of OpenAspectS. After discussing related work in Section 7, we summarize our paper in Section 8 and come to a conclusion.

1. Motivation

1.1. Aspect composition models

In most aspect-oriented systems there is typically a weaving mechanism that composes aspects and the base system they apply according to descriptions offered by pointcut predicates or expressions. Usually, such a composition is initiated by developers at a particular point in time. This point in time might vary from development-time, over compile- and load-time, up to runtime. Here it is important to note that each and every composition is either carried out implicitly by development tools or explicitly by instructions stated explicitly in the flow of control of the running system. This process can be characterized as one-time model composition.

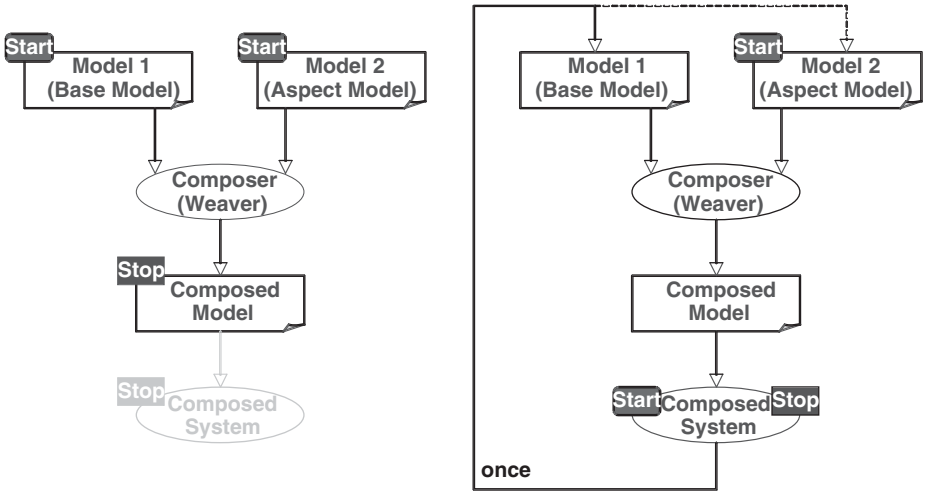

Fig. 1. (a) Static and (b) dynamic one-time composition.

Fig. 1a illustrates static one-time model composition. Weaving starts from both a model of the base system and the aspect model (as indicated by the 'start' tags). A composer then combines both these models into a composed model. For that, all join-point shadows involved are instrumented so that the composed model shows all desired properties (marked with a 'stop' tag). Next the composed model gets effective in the composed system at runtime. Note that in all figures showing composition models (Figs. 1,2,11) the dog-eared rectangles represent passive models, whereas the ovals represent active ones, that is, system parts being executed.

In Fig. 1b we can see an extension of the previously described process for dynamic one-time model composition. Now we start off from a running system and an aspect model. The weaver derives the base system's model from the running base system and then composes this derived and the aspect model into the composed model which in turn will be made effective in a new version of the running composed system. Since developers can initiate weaving at any point in runtime, we can treat the initial base system as a special case of the composed system.

Even though this procedure can be performed repeatedly, we still characterize it as one-time since the injection or removal of join-point shadows is set off by an explicit activity in the development process or program expression at load or runtime. In both cases of one-time weaving, the weaver only considers join-point shadows described by all involved pointcut expressions or predicates at that particular point in time. Future changes to the system that were not planned for, both to the base system and to the set of incorporated aspects, cannot be considered properly or at all.

Even with continuous weaving, presented in our work on Morphing Aspects [12], there are changes not taken into account. As shown in Fig. 2, morphing aspects constantly derive a minimal base model needed to inject or remove join-point shadows necessary in the immediate future.

While the weaver for every new step examines the current system, all previous system compositions are not revisited for changes that might have had an influence on those earlier compositions.

In the following we provide an example illustrating these effects.

1.2. Running example

Consider the following example as described in [10]. It is written in AspectS [10], a general-purpose dynamic AOP environment for Squeak/Smalltalk [13].[1] Squeak is an open and highly portable implementation based on the original

[1] Readers not familiar with Squeak/Smalltalk but articulate in Java might find the language syntax comparison in [14] of help when examining code fragments throughout the paper.

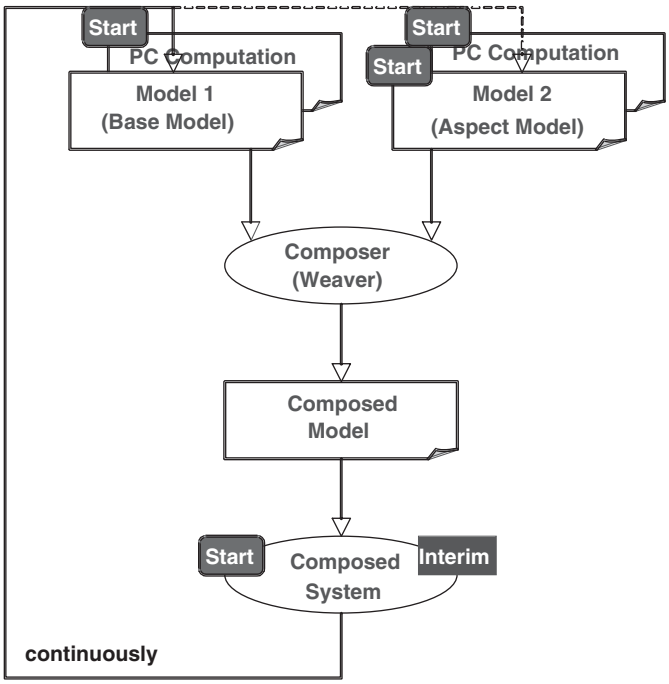

Fig. 2. Dynamic continuous composition.

```
MorphicMousingAspect>>adviceMouseEnter

    ↑ BeforeAfterAdvice
        qualifier: (AdviceQualifier
            attributes: { #receiverClassSpecific. })
        pointcut: [
            Morph withAllSubclasses
                select: [:m | m includesSelector: #mouseEnter:]
                thenCollect: [:m | AsJoinPointDescriptor
                    targetClass: m targetSelector: #mouseEnter:]]
        beforeBlock: [:receiver :arguments :aspect :client |
            Transcript show: '*Enter*', arguments first printString]
```

```
"aspect lifecycle in a nutshell"
| anAspect |
anAspect ← MorphicMousingAspect new.
anAspect install.
anAspect uninstall.
```

Fig. 3. Advice and lifecycle example.

Smalltalk-80 system [15]. Here we want to monitor all mouseEnter: and mouseLeave: messages received by instances of Morph (a base class in Morphic, a user interface framework of Squeak) and its subclasses by logging them to the system transcript, Smalltalk's equivalent to a system console (Fig. 3).

We employ an aspect called MorphicMousingAspect to trace the reception of these messages. Advice code to trace the reception of mouseEnter: and mouseLeave: messages is stated in two advice methods adviceMouseEnter and adviceMouseLeave. Each advice method creates a BeforeAfterAdvice object that allows us to state behavior before and after the invocation of a method. Once the advice object is created, it is further qualified via the #receiverClassSpecific advice qualifier attribute causing the advice code to be executed for all message receivers described by the pointcut

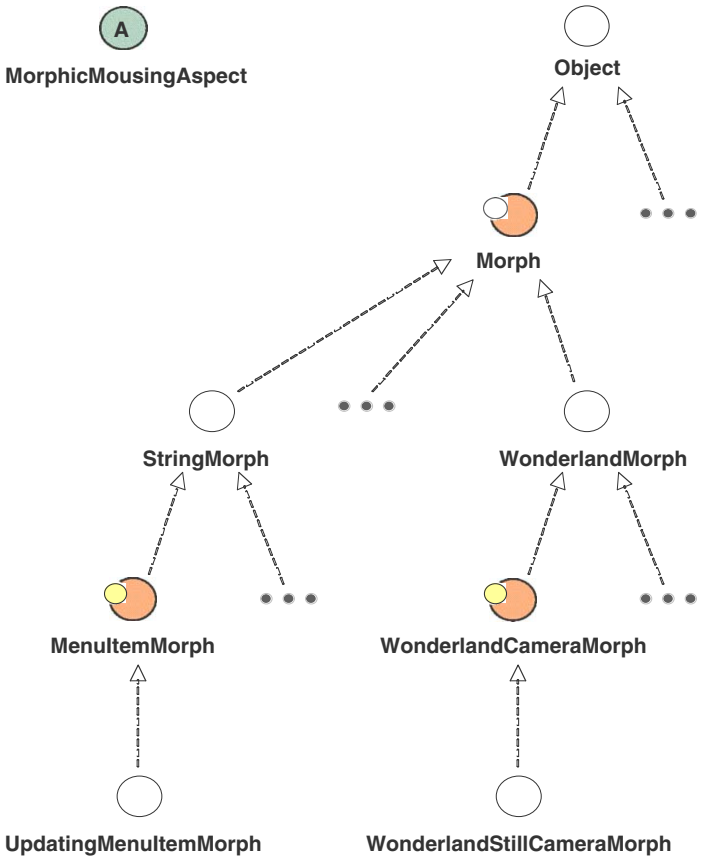

Fig. 4. Visualization of class hierarchy with installed aspect.

expression. In our example from Fig. 3, these are all instances of Morph and its subclasses responding to mouseEnter:. In adviceMouseEnter join-point descriptors are collected by querying the system for all classes that are subclasses of Morph and implement mouseEnter:. The block to be executed before the actual invocation of mouseEnter: echoes the event passed with the mouseEnter: message to the transcript. An adviceMouseLeave advice works likewise for the reception of mouseLeave: messages. To activate the MorphicMousingAspect, one creates an aspect instance and sends it an install message when desired. An installation activity can be considered atomic. In a plain Squeak image (version 3.6) there are 24 implementers of a method named mouseEnter: and 21 implementers of a method named mouseLeave:. 22 of the 24 of mouseEnter: methods and 19 of the 21 of mouseLeave: methods are found in Morph and its subclasses. Our instance of MorphicMousingAspect, when installed, instruments all $22 + 19 = 41$ locations.

Fig. 4 shows a subset of the Morph class hierarchy emphasizing some of the classes affected by the installation of MorphicMousingAspect by rendering them with a squared texture. All classes with behavior not influenced by the installation of the aforementioned aspect are displayed as circles with a white filling area. Little circular symbols on top of circles representing classes mark them as holding join-point shadows that belong to the pointcut of the aspect under consideration. Here, Morph, MenuItemMorph, and WonderlandCameraMorph are marked as being involved in MorphicMousingAspect.

1.3. Changing the base system

What happens if, after the installation of MorphicMousingAspect as described above, a new class MouseEnterLeaveMorph is added to our system (Fig. 5)? MouseEnterLeaveMorph is a subclass of Morph and also re-implements mouseEnter: and mouseLeave:. By doing so, this class would have been part of the set of join-point descriptors

```
Morph subclass: #MouseEnterLeaveMorph
    instanceVariableNames:''
    classVariableNames: ''
    poolDictionaries:''
    category: 'AspectS-Examples'

MouseEnterLeaveMorph>>handlesMouseOver: evt
    ↑ true

MouseEnterLeaveMorph>>mouseEnter: evt
    self beep Primitive.

MouseEnterLeaveMorph>>mouseLeave: evt
    self beep.
```

Fig. 5. Code of newly added Morph subclass.

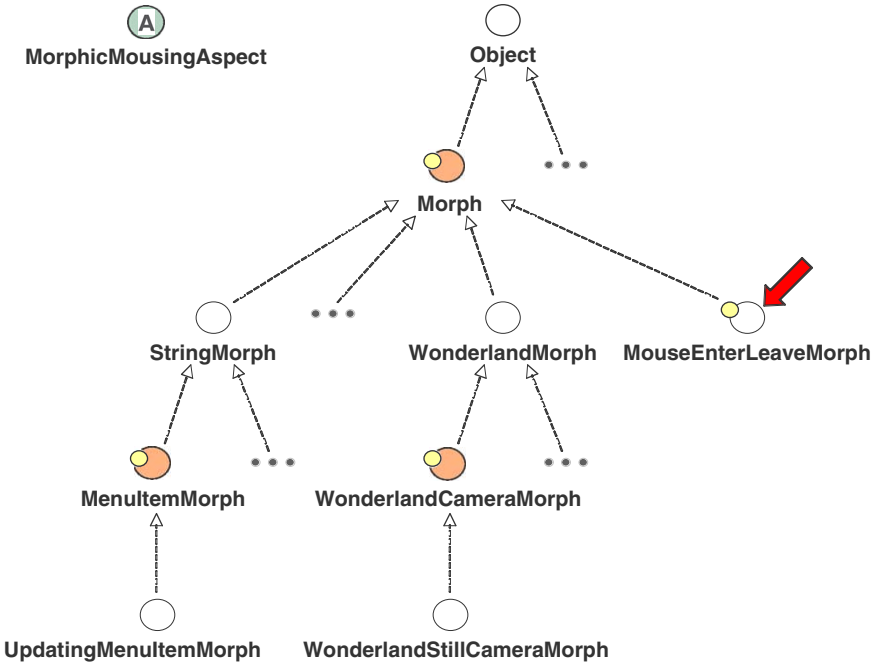

Fig. 6. Visualization of aspect composition after the addition of the new subclass of Morph.

computed by the evaluation of our pointcut expressions and instrumented by our weaver. However, when our aspect was installed this class was not yet present in our system and was thus not considered during weaving.

The result of adding this class after installing our aspect is shown in Fig. 6. Even though the evaluation of the pointcut expression would include the newly added class and its mouseEnter: and mouseLeave: methods (as indicated by the little circular symbol added to the circle representing MouseEnterLeaveMorph), the class as such remains unaffected by our aspect (as suggested by the white fill color of the class symbol). Our installed aspect only covers the 41 locations computed during weaving, leaving the new additional two locations unaffected.

Depending on specific application scenarios, these two missing join-points shadows might or might not cause erratic system behavior. Leaving them unaffected would honor the intent to only weave-in all join-point shadows covered by the aspect's join-point expression at installation time. On the other hand, adjusting them automatically would ensure the consistent application of the aspect to all join-point shadows covered by its join-point expression.

The described change scenario is an additive one. Subtractive changes to the base system have similar effects.

```
...
pointcut: [
    StringMorph with All Sub classes
        select: [:m | m includesSelector: #mouse Enter:]
        thenCollect: [:m | As Join Point Descriptor
            target Class: m target Selector: # mouse Enter:]]
...
```

Fig. 7. Changed pointcut expression of installed aspect.

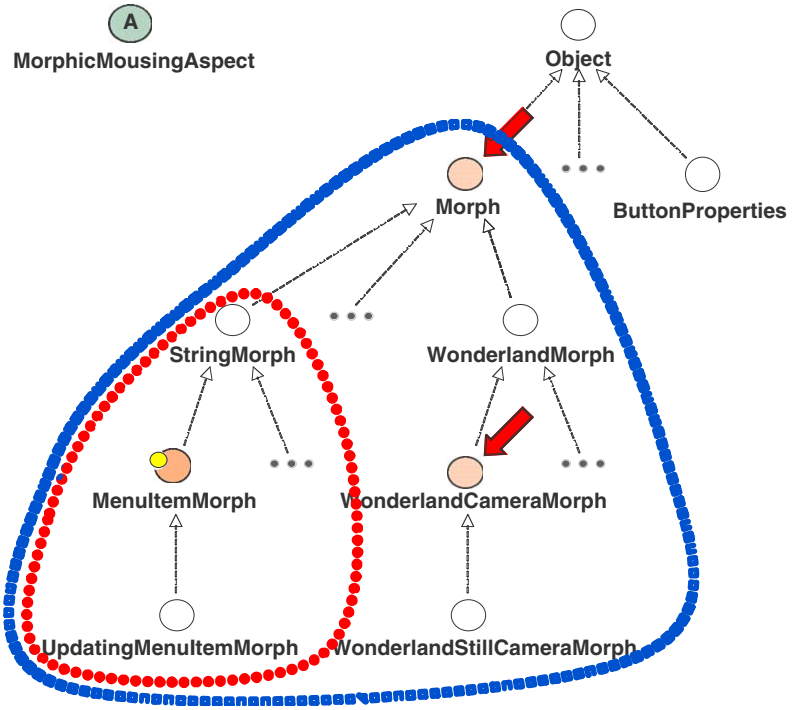

Fig. 8. Visualization of composition after changed pointcut coverage.

1.4. Changing an advice's pointcut

What happens if, after the installation of MorphicMousingAspect as described above, the pointcut expression of the installed aspect is changed? What if the pointcut expression of MorphicMousingAspect is, for example, modified to collect all join-point descriptors by querying the system for all implementers of mouseEnter: and mouseLeave: at StringMorph which is a subclass of Morph instead at Morph itself (Fig. 7)?

Changing our pointcut expression in such a way reduces the number of join-point descriptors contained in the set it evaluates to from 22 to one for mouseEnter:, and from 19 to one for mouseLeave: (Fig. 8). This leads to the same situation as described above where our pointcut became out of sync with the system actually instrumented in the process of installing the new class MouseEnterLeaveMorph.

Depending on specific application scenarios, the 39 join-point shadows that are still in the system but no longer covered by the aspect's pointcut expression might or might not cause erratic system behavior. Leaving them in the system would honor the intent to only weave-in join-point shadows covered by the aspect's join-point expression at installation time. On the other hand, removing them automatically would ensure the consistent application of the aspect to all join-point shadows covered by its join-point expression.

```
...
beforeBlock: [:receiver :arguments :aspect :client |
    Logger count increment.
    Transcript show: '*Enter*', arguments first printString]
...
```

Fig. 9. Changed advice code of installed aspect.

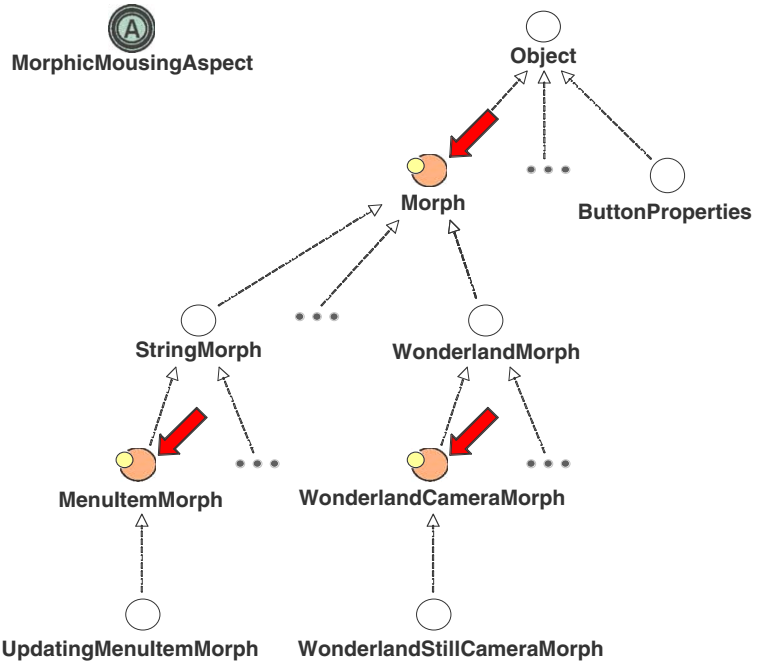

Fig. 10. Visualization of composition after changes to advice code.

1.5. Changing an aspect's advice

Also, consider what happens if, after the installation of MorphicMousingAspect, the advice code gets changed (Fig. 9)? The weaver previously used the advice code present during weaving to instrument the system. In our example, the weaver provisions advice code that logs the execution of mouseEnter: and mouseLeave: methods to the system transcript.

Changing that particular code located within our advice block leaves all 41 locations in our image that are instrumented with the previously available advice code out of sync with its current version. In Fig. 10 the different versions of the advice code residing in the aspect and composed into our system are expressed by different line styles of the circles representing our aspect and system classes.

Again, correct system behavior depends on particular application scenarios that might require the instrumented system parts to be updated or left as is. Instead of leaving the actual adaptation behavior to a particular weaving mechanism, we prefer to give the developer a means to explicitly decide about adaptations if demanded.

2. Open Aspects

2.1. System changes

In open systems that are allowed to change at runtime, aspect composition needs to explicitly address changes to both the base system and the set of aspects that have been applied to it. These changes comprise added,

transformed, or removed classes and methods. The addition, modification/transformation, or removal of pieces of advice associated with an aspect or pointcut expressions or predicates associated with a piece of advice also need to be handled.

A weaver determines all join-point shadows of all involved aspects by evaluating all associated join-point expressions or predicates. After completion, the set of join-point shadows available in the composed system correspond to the ones computed during weaving. Every change to the system has the potential to bring this correspondence out of sync. If not dealt with correctly, such inconsistency may or may not lead to erratic system behavior.

2.2. Change events and corrective actions for open aspects

The abovementioned changes leading to composition inconsistencies can be addressed in quite different ways. Whether or not corrective actions taken to react on system change events will lead to expected system behavior can only be decided in a particular application context. While some applications do not expect changes to be considered after the composition of aspects, others might be required to adjust aspect composition accordingly. Here, we will describe corrective actions we consider important and feasible for operating open systems in use:

- None/indifferent.
- Full reinstall.
- Partial reinstall.
- Partial withdrawal.
- Full withdrawal.
- Reject.

'None/indifferent', not reacting to any change happening after the installation of an aspect, is the simplest corrective action that may be taken. 'Full reinstall' will cause the affected aspect with all its pieces of advice to be uninstalled and then and reinstalled again afterwards, fully applied to the newly inserted set of join-point shadows. 'Partial reinstall' will reinstall only pieces of advice affected by a change, leaving all unaffected advice installed and untouched. 'Partial withdrawal' will uninstall only pieces of advice affected by a change, leaving all unaffected advices installed and untouched. 'Full withdrawal' will cause all affected aspects with all their pieces of advice to be withdrawn from the system for good. 'Reject' will prevent attempts to change the base system affected by respective advice directives to get effective. While these corrective actions seem the most obvious to us, there are certainly other that could be added to this list.

2.3. Adaptation models of Open Aspects

Adaptation models of Open Aspects are a means to explicitly provide corrective actions to be carried out in response to system change events. With adaptation models, Open Aspects allow for the flexible association of change events and corrective actions, according to specific needs of the application scenario to be supported and the system behavior to be achieved.

With Open Aspects there is an explicit separation of base, aspect, and adaptation models. This allows an explicit association of elements belonging to them. Such association expresses which elements of the base system need to be affected by which aspects and their associated pieces of advice under the occurrence of which set of change events. This lets us directly say how or if at all to react to system events. While this approach can certainly be extended to deal with any type of event, we will for now limit ourselves to system change events as described above.

2.4. Conditional weaving with Open Aspects

We characterize the weaving model of our approach to Open Aspects as dynamic conditional model composition. As illustrated in Fig. 11 there is now, besides the base and the aspect model, also an adaptation model to be considered by the weaver when composing such models. As already described for dynamic one-time composition (in Fig. 1b) and dynamic continuous composition (Fig. 2), we start from a composed system at runtime. The weaver initially derives a

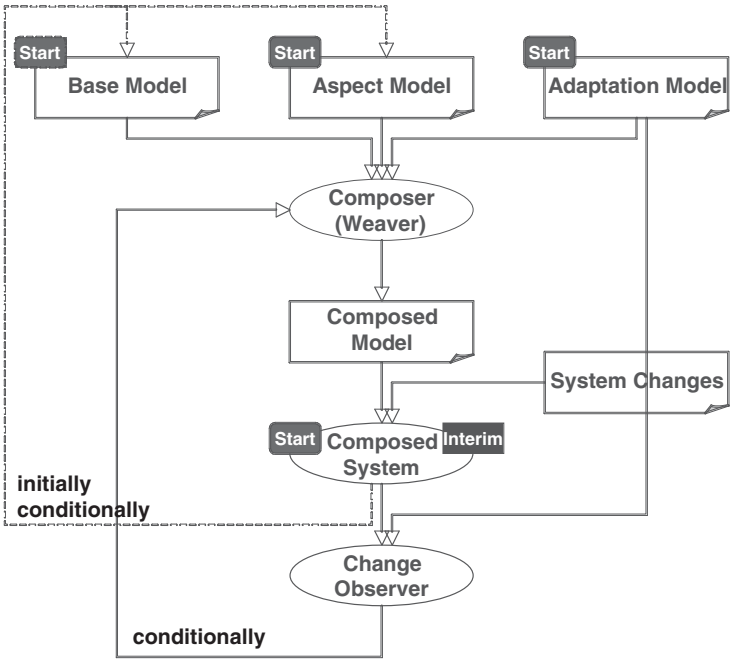

Fig. 11. Dynamic conditional composition.

model of the running base system needed for making our aspect model effective (both marked with a 'start' tag). While doing so, the weaver also examines an adaptation model (also marked with a 'start' tag) detailing all involved system change events to be observed and all corrective actions to be taken in correspondence to the system elements involved as described above.

The weaver affects the composed system as in the other dynamic models described before. In addition to that it provides for change event handling. Such event handling can be supported in a variety of ways. One way to address system change events is the inlining of event handling code into the composed system that itself initiates specified corrective actions (amongst them system re-composition) in the event of change. Another way is the provisioning of a separate system entity we call a change observer that reacts to system change events appropriately. Compared with the previous method, the composed system does not include any code related to conditional model composition. Event propagation as such may be implemented in quite a few ways. Implementations can vary from a very simple and naïve propagation from an event source to an event consumer to quite sophisticated event filters preprocessing events according to complex rule sets.

3. Illustration

3.1. Starting situation

In the following we will illustrate how Open Aspects behave in open systems. For that we will use the notation as shown in the right side of Fig. 12. In part (a), the left side of Fig. 12, we start out with two modules running on our platform with no aspects yet applied. This set of modules can be extended by adding new modules or curtailed by removing existing ones.

The L-shaped open platform represents that part of the runtime environment that remains quite stable over time with respect to change. Boxes represent modules that are expected to change. If affected by an open aspect, such a box is displayed with squared pattern texture, or is gray otherwise. Open Aspects are rendered as ovals. If an open aspect was changed, its boundary appears as a zig zag line.

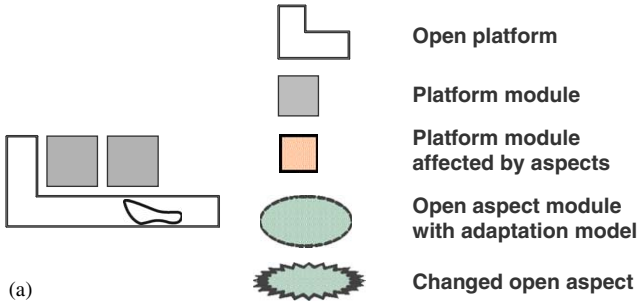

Fig. 12. Computational platform (with legend).

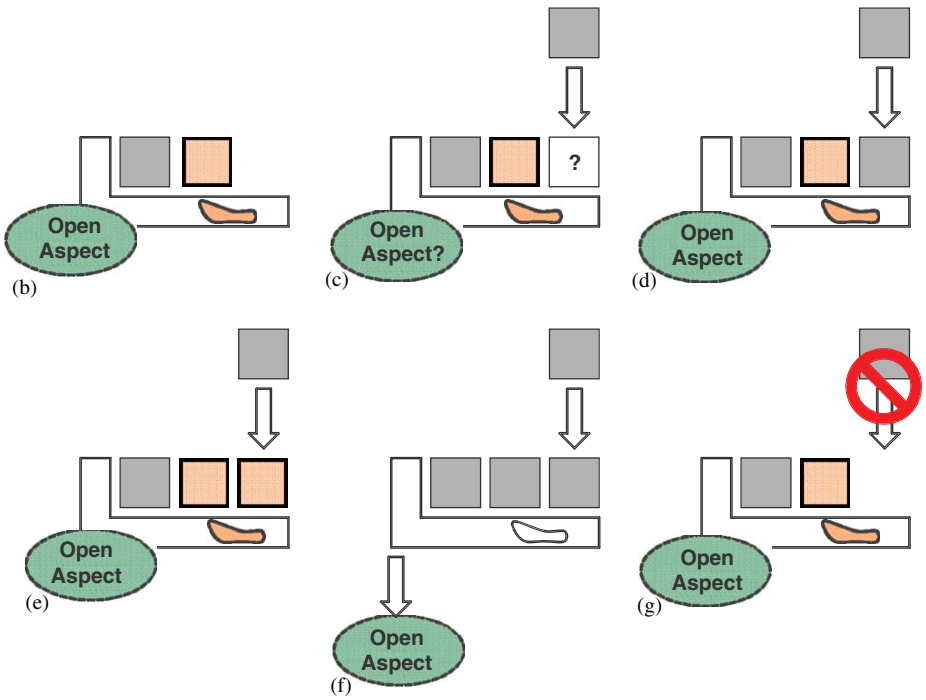

Fig. 13. Additive changes to the base.

3.2. Additive changes to the base

In Fig. 13 we demonstrate composition adaptation with respect to additive changes to the base system. In part (b) an open aspect gets applied to the system depicted in part (a) of Fig. 12.

This application affects one module (the second box from the left) as well as part of the platform. Note that the effect of an open aspect to the platform itself is only shown to emphasize that Open Aspects can be applied to the platform itself as well and are not limited the user-supplied modules. Part (c) presents an additive change to be dealt with by our Open Aspects infrastructure. Here a third component is put on our platform, with an open aspect already installed. Parts (d) to (g) illustrate how the system might respond to such change according to the adaptation model associated with the open aspect applied.

In (d), we can see the most simple of all adaptation strategies in action: ‘none/indifferent’ which leaves the system as is, without taking any corrective action. Note that in this case the same behavior can be observed if there is an adaptation strategy applied other than ‘none/indifferent’, but that under the given circumstance the adaptation model

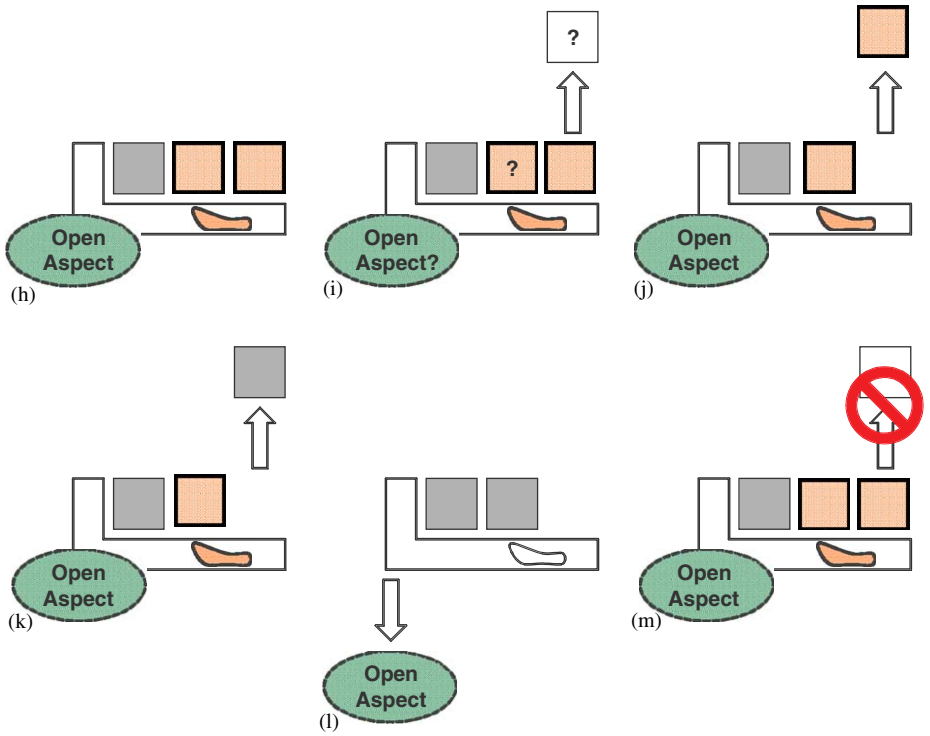

Fig. 14. Subtractive changes to the base.

does not require any corrective action to be taken. In (e), the newly added component requires some aspect-related composition and is adjusted accordingly. An adaptation model requiring a 'partial or full reinstall' could have instigated this behavior. The effect of executing a 'full withdrawal' can be seen in part (f). Part (g) shows that the Open Aspects infrastructure might as well refuse the addition of further components if required by an adaptation model.

3.3. Subtractive changes to the base

In Fig. 14 we explain composition adaptation in response to subtractive changes to the base system.

In (h) we start from a situation in which an open aspect applied to our system affects two of its three modules. We are now going to remove one module, as show in part (i). Parts (j) to (m) illustrate how the system might respond to such subtractive change according to the adaptation model associated with the open aspect applied.

In (j), the system is left as is, without taking any corrective action as a result of a 'none/indifferent' adaptation strategy. In (k) the removed component will be freed from all compositions by the applied Open Aspects that were effective previously. This, for example, can be the result of a 'partial withdrawal' or a 'full reinstall'. Reverting a removed module to the form it was in previously to its addition to the system can be of interest to modules that have to be moved to storage or into another execution context such as another system to enable a defined launch thereafter. The effect of executing a 'full withdrawal' as response to the subtractive change can be seen in part (l). Part (m) shows that the Open Aspects infrastructure might as well refuse the further removal of components if required by an adaptation model.

3.4. Aspect changes (pointcut and advice)

In Fig. 15 we show how changes to an open aspect itself might affect all compositions originated by this aspect so far, according to the adaptation model associated with the open aspect installed. The aspect installed in (n) affects two of the three modules available on our platform. What happens to the system if this aspect (some or all of the advice

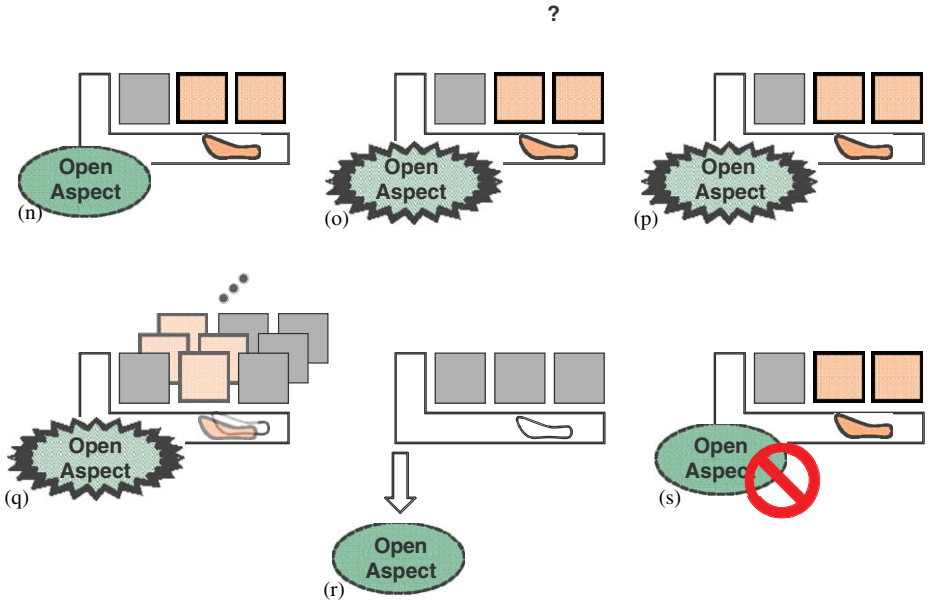

Fig. 15. Changes to the aspect itself.

code associated with it, or some or all pointcuts associated with its pieces of advice) is changed as indicated in (o)? Parts (p) to (s) describe corrective actions that could be taken to address such change to an open aspect.

In (p), no corrective action is taken at all, leaving our system in the composition state as observed before the change to the open aspect. In (q), several possible combinations of changes are implied to the modules or the platform as a result of a 'partial withdrawal', or 'partial or full reinstall' if required by the adaptation model associated with the changed aspect. Similar to additive or subtractive changes to the base system, changes to an open aspect might result to either a 'full withdrawal' (r) or the refusal of the change itself (s).

4. Implementation

4.1. AspectS

AspectS extends Squeak to allow for experimental aspect-oriented system development. Its goal is to provide a platform for the exploration of dynamic late-bound aspect-oriented software composition. It employs coordinated meta-level programming to address the tangled code phenomenon. AspectS shows great flexibility by not relying on code transformations, but by making use of metaobject composition instead. AspectS provides a framework for developers to construct the proper runtime structure of aspect instances. Once instantiated, an aspect instance refers to its associated advice objects that maintain all information about what additional code (Computation, an instance of BlockContext) has to be performed where (Pointcut, an instance of BlockContext, to compute all shadow join-points to instrument) and when (described via AdviceQualifier attributes).

Weaving or unweaving happens every time an install or uninstall message is sent to a respective aspect instance. Installation causes the pointcut computation to be executed returning a set of join-point descriptors indicating locations in the system structure to be affected. Then for each join-point descriptor the weaver creates an appropriate method wrapper [16] instance matching the advice type. Each such wrapper is then configured with the actual advice code as well as one or many so-called activation blocks. Activation blocks are selected according to advice qualifier attributes provided by the developers. They perform residual runtime tests deciding if the advice code a join-point shadow was instrumented with is going to be executed or not. Fig. 16 shows both the metaobject structure created by programmers using the AspectS framework, as well as the metaobject structure constructed or affected by the weaver when installing or uninstalling an aspect. Note that the latter object structure is based on the former.

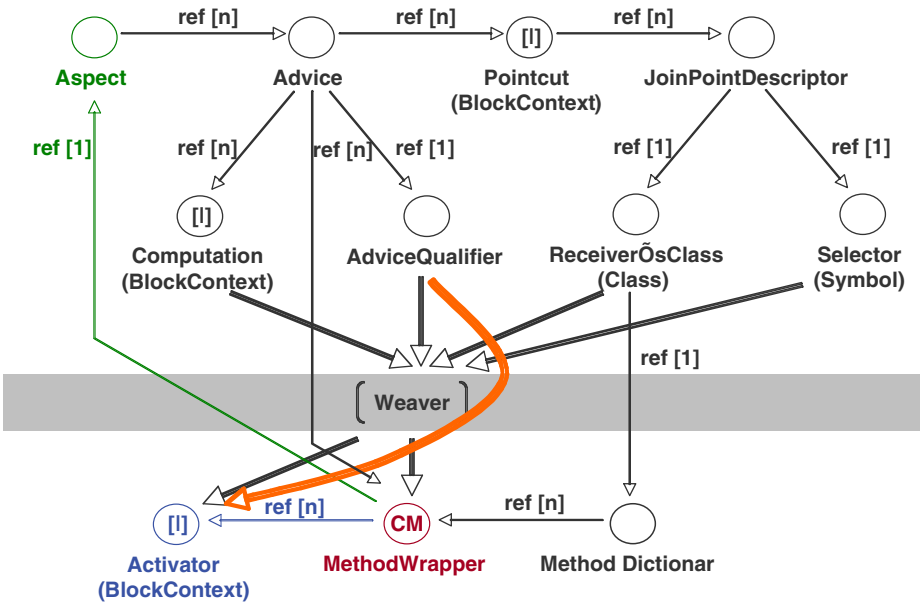

Fig. 16. Weaving in AspectS.

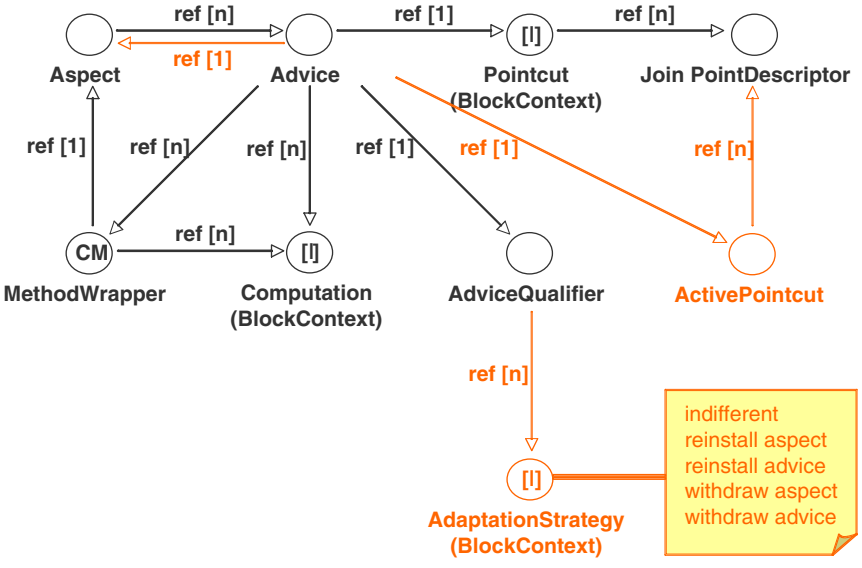

Fig. 17. OpenAspectS runtime structure extensions.

4.2. *OpenAspectS extensions*

OpenAspectS is our prototypical implementation of Open Aspects. It is an extension to AspectS. OpenAspectS is based on Squeak version 3.6 and AspectS version 0.5.4. OpenAspectS extends AspectS' basic runtime structure as shown in Fig. 17. We added an active pointcut (ActivePointcut) system element associated with each advice. An active pointcut object records the set of all join-point descriptors that were associated with that aspect when the installed aspect was woven into the system. This set of join-point shadows is obtained by executing the pointcut expression (Pointcut) associated the respective advice, as described above.

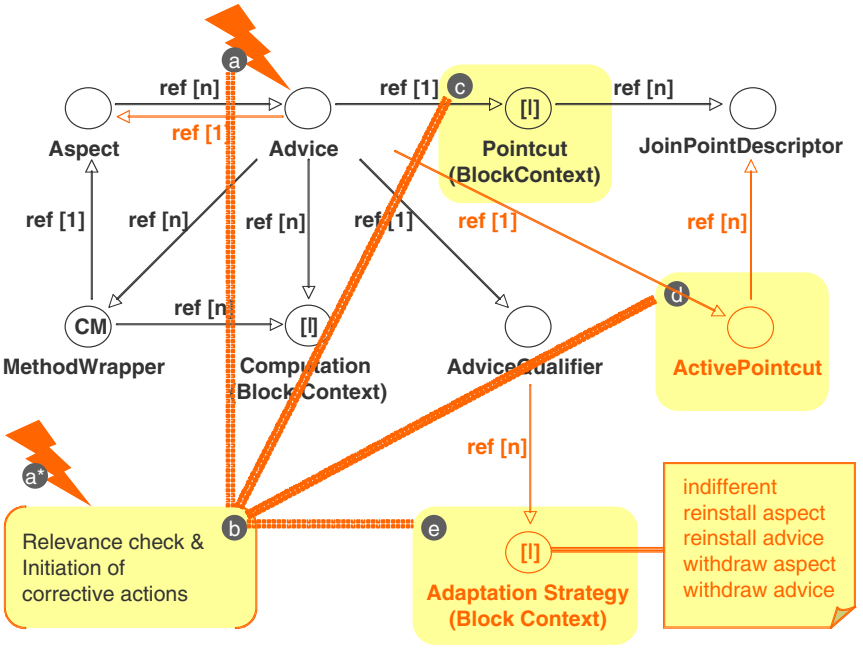

Fig. 18. OpenAspectS change events and relevance checks.

In OpenAspectS, we added adaptation strategies (AdaptationStrategy) to advice qualifiers (AdviceQualifier) implementing corrective actions to be taken in the presence of change relevant to a particular advice or its associated aspect. In our current implementation we provide the following adaptation strategy examples:

- None/indifferent (via the #indifferent advice qualifier adaptation attribute).
- Full reinstall (via the #reinstallAspect advice qualifier adaptation attribute).
- Partial reinstall (via the #reinstallAdvice advice qualifier adaptation attribute).
- Partial withdrawal (via the #withdrawAdvice advice qualifier adaptation attribute).
- Full withdrawal (via the #withdrawAspect advice qualifier adaptation attribute).

The extended advice qualifier allows us to explicitly associate adaptation strategies with advice and aspects.

According to the specified adaptation attributes, the weaver selects the appropriate adaptation strategies and initiates the registration of the aspect or advice instance with the change notification infrastructure.

4.3. Changes and relevance checks

In order to make Open Aspects aware of system change events of interest, we extended Squeak's base to provide us with change notifications for each change to a class or its methods. We modified Squeak to provide proper system change events, as well as with a single point of registration for such notifications. System change notifications are now supplied for each addition, transformation, or removal of individual classes or methods, similar to the dependent maintenance protocol of the CLOS Metaobject Protocol (MOP [17]). Furthermore, we provide such notifications right before and right after one of the aforementioned changes are carried out by the system. This gives us more flexibility and more accuracy in selecting suitable corrective actions.

We implemented a mechanism to determine if a change to the system indicated by a system change event is relevant to an individual advice or aspect (Fig. 18).

When an aspect and all of its advice is installed, each advice registers for system change events listed above. If there is a change event, an advice that expressed interest is informed (a). In response to that, the determination of relevance is started (b). For this relevance check the pointcut expression is reevaluated (c). The newly computed result representing

```
MorphicMousingOpenAspect>>adviceMouseEnter

↑ BeforeAfterAdvice
    qualifier: (AdviceQualifier
        attributes: { #receiverClassSpecific. }
        adaptations: { #reinstallAdvice. })
    pointcut: [
        Morph withAllSubclasses
            select: [:m | m includesSelector: #mouseEnter:]
            thenCollect: [:m | AsJoinPointDescriptor
                targetClass: m targetSelector: #mouseEnter:]]
    beforeBlock: [:receiver :arguments :aspect :client |
        Transcript show: '*Enter*', arguments first printString]
```

Fig. 19. Example advice in OpenAspectS.

the set of join-point shadows that would be provisioned if the aspects would be installed now is then compared with the actual set of join-point shadows (stored as active pointcut) the aspect was associated during its actual installation (d). If the comparison indicates that the two sets are different and the adaptation model requires a corrective action to be taken, a corresponding adaptation strategy is selected and executed (e). Note that this is a simplified implementation to illustrate the basic flow of events. More complex systems might demand more complex and sophisticated event subscription and distribution mechanisms.

4.4. More implementation options

In another implementation we allowed the developer to explicitly associate change event types and corrective actions by accepting pairs of change and action descriptors as adaptation attributes in advice qualifiers. Examples of such pairs are #(changedAdvice fullReinstall), #changedPointcut partialReinstall), and #(changedMethod indifferent). In our current implementation we decided, for pragmatic reasons, to treat all change events evenly and because of that we allow only action descriptors to be given as adaptation attributes in advice qualifiers.

5. Application

5.1. Starting situation

Our Open Aspects example starts out similarly to the one described in Section 2. This time we are utilizing the Open Aspects platform as shown in our code example. The main difference for the programmer is the additional adaptations attribute section for advice qualifiers.

In Fig. 19 we can see that the developer decided to let the advice being reinstalled in the event of a change to the base system that extends to the span of this piece of advice (adviceMouseEnter).

Since the pointcut expressions of adviceMouseEnter (as seen in Figs. 3 and 19) and adviceMouseLeave are the same as of MorphicMousingAspect, the activation of an instance of MorphicMousingOpenAspect instruments the same 41 locations in the image as well (Fig. 20).

5.2. Changing the base system

With an open aspect instance of MorphicMousingOpenAspect installed in our system, adding the new class MouseEnterLeaveMorph implementing mouseEnter: and mouseLeave: will change our system as shown in Fig. 21. Here the part of the class hierarchy framed with a box labeled *previously* denotes the observable effect prior to Open Aspects.

Our reinstall-advice adaptation strategy, selected in our advice qualifier by providing the #reinstallAdvice adaptation attribute, caused the Open Aspects environment to notice the addition of a new class, its having an effect to the pointcuts of adviceMouseEnter and adviceMouseLeave of MorphicMousingOpenAspect. As a response to this change, the two pointcut expressions are reevaluated and the pieces of advice associated with them are reinstalled.

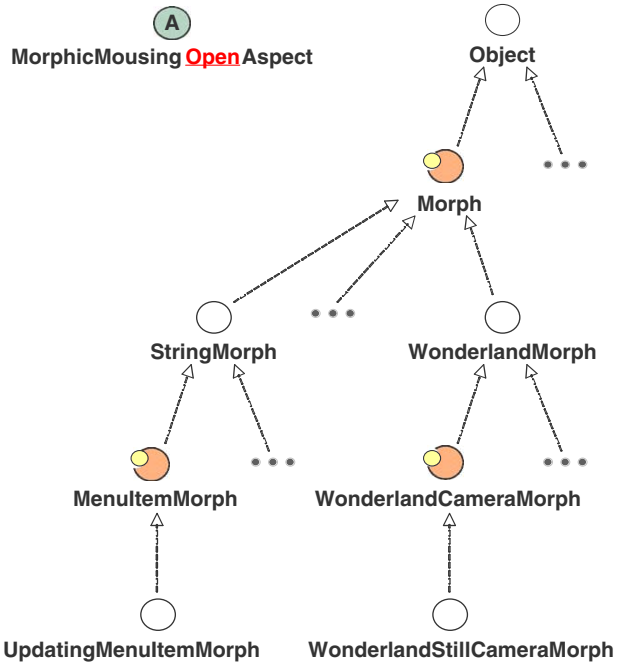

Fig. 20. Morph sub-hierarchy with installed open aspect.

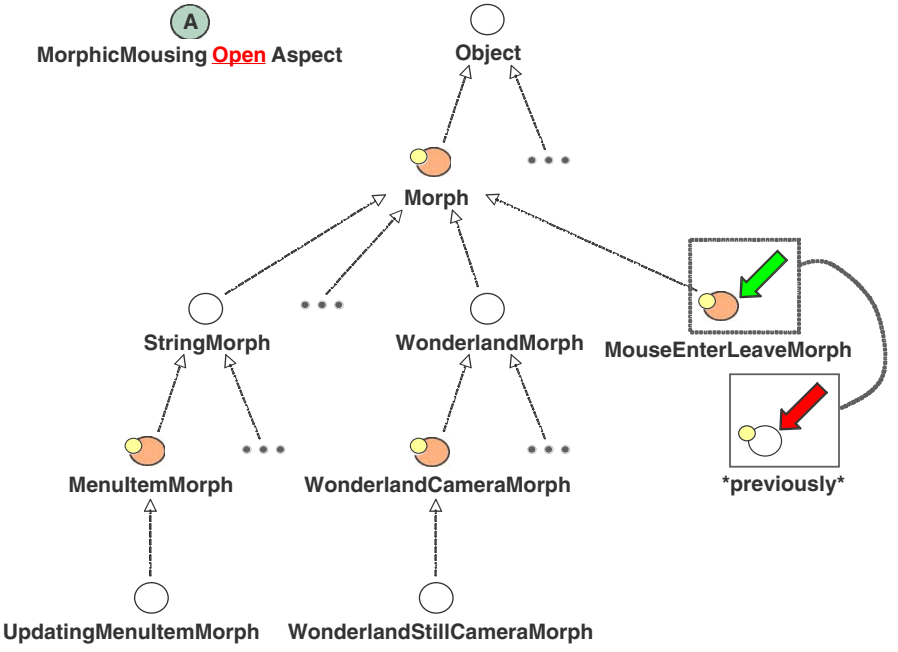

Fig. 21. Visualization of composition after class addition.

Besides reacting properly on additive changes, transformative and subtractive changes need to be addressed appropriately as well. In the following we show how the removal of methods and classes covered by the pointcuts of our installed aspect instance of MorphicMousingOpenAspect can affect the aspect composition.

```
MouseEnterLeaveMorph removeSelector: #handlesMouseOver:.
MouseEnterLeaveMorph removeSelector: #mouseEnter:.
MouseEnterLeaveMorph removeSelector: #mouseLeave:.

WonderlandCameraMorph removeSelector: #handlesMouseOver:.
WonderlandCameraMorph removeSelector: #mouseEnter:.
WonderlandCameraMorph removeSelector: #mouseLeave:.
```

Fig. 22. Code removing mouse enter and leave methods.

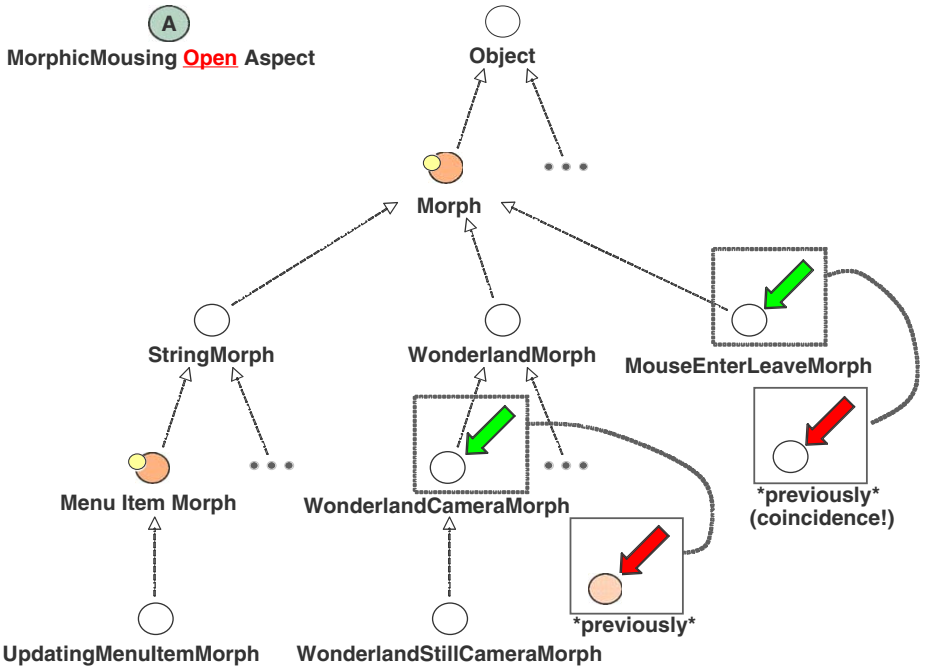

Fig. 23. Visualization of composition after method removal.

In Fig. 22 we see code that removes the code to handle mouse-enter and mouse-leave events from the two classes MouseEnterLeaveMorph and WonderlandCameraMorph. The resulting composition that is based on our reinstall-advice adaptation policy is illustrated in Fig. 23.

Because of the removal the set of join-point shadows to be instrumented by the installation of the MorphicMousing OpenAspect has changed. This change caused our adaptation strategy to reinstall our aspect that now does not affect the aforementioned join-point shadows anymore.

5.3. *Changing an advice's pointcut*

Changing an advice's pointcut expression while instances of such aspect are active might also need to be addressed on a case-by-case basis. If deliberately ignored or accidentally overlooked, the example of a changed pointcut, as listed in Fig. 7, can leave our system in a state displayed in Fig. 8.

The selection of an indifferent adaptation strategy for an advice of an open aspect would have lead to the same result. An adaptation strategy to reinstall all affected pieces of advice or the aspect they are associated with yields the change of aspect composition as illustrated in Fig. 24: Since mouseEnter: and mouseLeave: of Morph and WonderlandCameraMorph are not covered by the new version of our pointcut expression anymore (as indicated by the missing dots that have marked them previously), our aspect composition is revoked from there also.

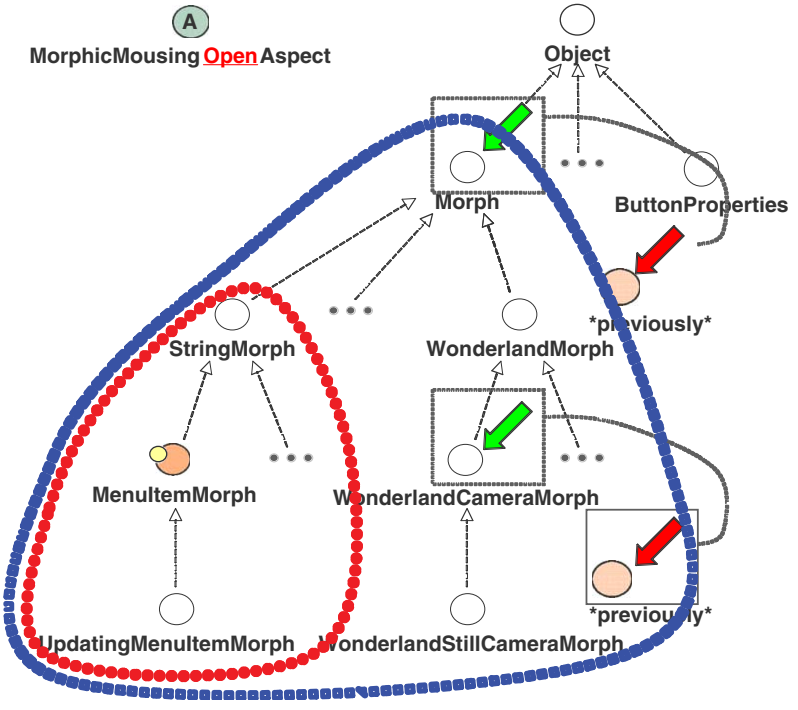

Fig. 24. Visualization of composition after change to pointcut coverage.

5.4. Changing an aspect's advice

Changing the advice code of a composed aspect might cause defective system behavior as well. If deliberately ignored or accidentally overlooked, the example of changed advice code, as listed in Fig. 9, can leave our system in a state displayed in Fig. 10. The selection of an indifferent adaptation strategy for an advice of an open aspect would have lead to the same result.

An adaptation strategy to reinstall all affected pieces of advice or the aspect they are associated with yields the change of aspect composition as illustrated in Fig. 25. Since the before blocks of adviceMouseEnter and adviceMouseLeave were changed, the composition of the two pieces of advice were adjusted as well to the new behavior (as indicated by the different style of the border of the class symbols the composition is made effective).

6. Related work

6.1. Load time weaving

Class loading in Java represents a simple form of open systems. Hence, if a system is designed in a way that it uses classes that are not known at compile-time then the underlying system is open. JMangler [18], Javassist [19], and EuLisp [20] are systems that permit the adaptation of classes at runtime. Hence, it permits one to adapt system changes (namely the addition of new classes to a running system). From the technical perspective, these systems would allow to build up an adaptation model based on the loading-a-new-class system event. However, the possible actions of the adaptation model are quite restricted due to the underlying language design: while the adaptation of the newly loaded classes can be easily achieved, the deletion of adapted join point shadows is not possible for classes which are already instantiated.

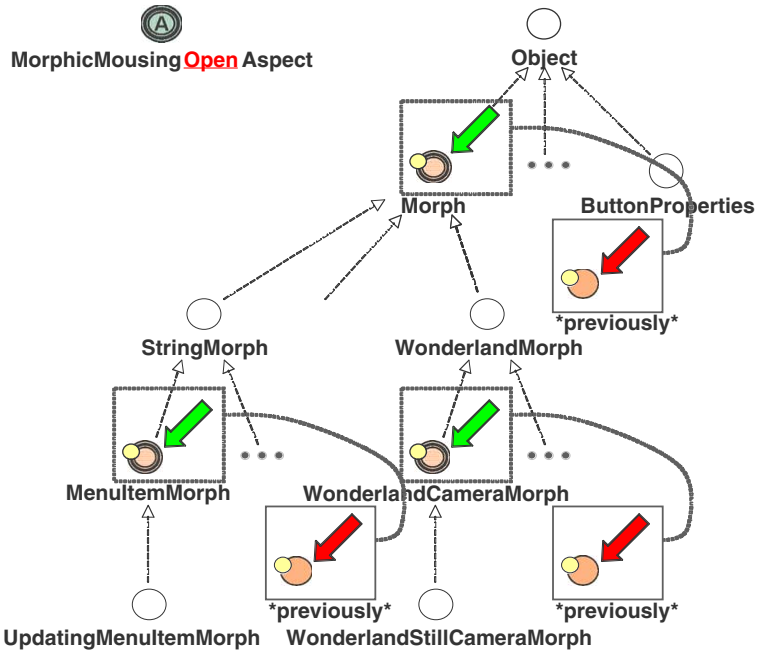

Fig. 25. Visualization of composition after change to advice.

6.2. Dynamic weaving via interpretation

There are already a number of systems that permit one to weave aspects dynamically based on the interpretation of the underlying code base. The most often mentioned system is PROSE [5,6], which is based on the programming language Java. It extends the underlying runtime system to invoke aspect-specific code when a certain join-point is reached by utilizing break points of the Java debugging interface. On an abstract level, this means that specific events (for example method or constructor calls) are redirected: for a given call it is checked to see whether aspect-specific code needs to be executed. From that point of view, they potentially permit weaving to react on changes of the underlying system: changes of the base system could be noted by having a corresponding join-point on the method that is responsible for the change. Due to the limitations of Java in respect to changes of the underlying system, however, the problems that typically arise in open systems rarely occur here: classes or single methods are not (or rarely) considered to change at runtime. In contrast to static weaving classes that were not present at compile time, but which are dynamically loaded, may also be considered by woven aspects. An explicit adaptation model as proposed in this paper is not considered.

6.3. Dynamic weaving via adaptation

A number of systems achieve dynamic weaving via code adaptation. Systems like Steamloom [7], AspectS, or the Selective Just-in-time Weaver [21] also permit dynamic weaving, but in contrast to the previously mentioned approach they adapt the underlying code base. Although they technically work quite differently, they have one point in common: at one point in time (when an aspect is woven) there is a computation that determines all join-point shadows [11] of a certain aspect and adapts these join-point shadows in some way. Loading a new class to the system differs from the previous approach. Loading a new class that was not available when the pointcut computation was done would not be considered by the corresponding aspects. Hence, such systems also suffer from the problem of open systems.

In AspectL [9], an aspect-oriented extension of CLOS [22], the concept of generic pointcuts is introduced that allow adding methods on the one hand and aspect weavers on the other hand. Whenever a method is added to a generic pointcut, all aspect weavers are triggered to generate new corresponding before/after/around methods. Likewise, whenever a new aspect weaver is added, it is applied to each already existing method in that pointcut. In AspectL, aspect weavers are not declarative but operational, making it necessary to use functions of the CLOS MOP [17]. AspectL does not

provide a declarative pointcut language. In other words, AspectL provides some basic machinery to keep aspects and base code in sync, which can be understood as a first step towards Open Aspects, but does not provide a full-fledged solution as described in this paper.

In AspectS weaving is achieved by instrumentation of instances residing in the Smalltalk meta-object structure. Since this structure changes permanently during development and runtime, the problem addressed by Open Aspects occurs frequently. However, previous versions of AspectS did not take these problems into account.

6.4. Continuous weaving

Morphing aspects as described in [12] permit one to weave aspects depending on the application's behavior. In particular, an aspect does not adapt all join-point shadows [11] where aspect-specific behavior is potentially needed upfront, but adapts a minimal set of join-point shadows. Additionally, it provides a computation strategy that describes how and when additional join-point shadows need to be adapted. As a consequence, weaving is not performed in a single step, but continuously during an application's runtime. However, the problem of changes in an open system is not addressed by morphing aspects: the adapted join-points where additional weaving needs to be performed are join-points of the underlying application, and not join-points of the underlying system. For example, the creation or deletion of a class is not considered as a join-point of Morphing Aspects in [12].

6.5. Incremental weaving

As a general-purpose AOP extension to IBM VisualAge for Smalltalk, Apostle [23,24] offers an incremental weaving mechanism to make the development of aspects fit better into the interactive and exploratory development approach of Smalltalk. It allows code to be added or modified over time. Changes affecting aspect compositions cause the weaver to incrementally adjust these compositions. Its incremental weaver, however, does not allow one to modify its behavior but reacts to changes always the same way.

6.6. Consistency maintenance

SmartTalk [25] implements the ability to keep the evolution of classes consistent with specified contracts related to subclassing semantics, base module properties, and the protection of implementation internals. In SmartTalk class changes are monitored and potential conflict situations such as accidental method capture or inconsistent methods are detected. Then, classes responsible for a particular conflict situation are informed, and automatic transformations handling such conflict are processed. Similar to Open Aspects, this approach is concerned with consistency in an open environment. Both SmartTalk and Open Aspects address changes to the underlying system in order to deal with inconsistencies. While SmartTalk is only concerned with changes to the bases system but not the transformations, Open Aspects deals with changes to both the base and the transformation system.

7. Summary and conclusion

In this paper, we identify the need to handle the weaving of aspects in open systems, that is to say systems that are intentionally designed to change at runtime, in a special way: it is necessary that the developer specifies how aspect compositions are to be maintained if changes to the system might affect them.

Open Aspects allow a system to respond to system change events appropriately by providing adaptation models. Adaptation models explicitly associate such change events with corrective actions to be taken in the event of change-change to the base system or to aspect compositions themselves. Additive, transformative, and subtractive changes to regular objects and aspects with their methods, fields, pieces of advice, and pointcuts are dealt with based on dynamic conditional weaving.

Open Aspects can be applied by developers and maintainers to make changes and their effects to pointcuts, aspects, pieces of advice, and the compositions' base system visible. They help to indicate inconsistencies between intended and actual compositions over time and provide means to take corrective actions if desired and necessary.

OpenAspectS is our implementation of Open Aspects in Squeak/Smalltalk. It extends AspectS' runtime structure to become aware of system changes and to perform relevance checks upon which it is decided if a particular change needs to be addressed by an adaptation strategy or not at all.

Acknowledgements

We would like to thank Alexandre Bergel, Gilad Bracha, Pascal Costanza, Stephane Ducasse, Jeff Eastman, Erik Ernst, and Dave Thomas for their valuable discussions and contributions.

References

[1] Hirschfeld R, Kawamura K. Dynamic service adaptation. In: Proceedings of the ICDCS 2004 workshop on distributed auto-adaptive and reconfigurable systems (DARES), March 23–24, Tokyo, Japan: IEEE Press, 2004. p. 290–7.
[2] Hirschfeld R, Kawamura K, Berndt H. Dynamic service adaptation for runtime system extensions. In: Battiti R, lo Cigno R, Conti M, editors. Wireless on-demand network systems. Proceedings of WONS2004, Lecture Notes in Computer Science, vol. 2928. Springer; 2004. p. 225–38.
[3] Kawamura K, Hamard J, Hirschfeld R, Minokuchi A, Souville B. Sustainable evolutionary systems. In: NTT DoCoMo Technical Journal, vol. 6, no. 1, pp. 14–19, June 2004 (Japanese version appeared in NTT DoCoMo Technical Journal, vol. 12, no. 1, p. 15–19, April 2004).
[4] Lämmel R. A semantical approach to method-call interception. In: Proceedings of the conference on aspect-oriented software development (AOSD), April 22–26, Enschede, The Netherlands: ACM Press, 2002, p. 41–55.
[5] Popovici A, Gross Th, Alonso G. Dynamic weaving for aspect-oriented programming. In: Proceedings of the conference on aspect-oriented software development (AOSD), April 22–26, Enschede, The Netherlands: ACM Press, 2002. p. 141–7.
[6] Popovici A, Gross Th, Alonso G. Just in time aspects. In: Proceedings of the Conference on Aspect-Oriented Software Development (AOSD), March 17–21, Boston, MA, US: ACM Press, 2003. p. 100–109.
[7] Bockisch C, Haupt M, Mezini M, Ostermann K. Virtual machine support for dynamic join points. In: Proceedings of the conference on aspect-oriented software development (AOSD), March 22–26, Lancaster UK: ACM Press, 2004. p. 83–92.
[8] Pawlack R, Seinturier L, Duchien L, Florin G. JAC: A flexible solution for aspect-oriented programming in Java. In: Proceedings of reflection 2001, Lecture Notes in Computer Science, vol. 2192, Berlin: Springer, 2001. p. 1–24.
[9] Costanza P. A short overview of AspectL. In Proceedings of the European interactive workshop on aspects in software (EIWAS), Berlin, Germany, September 23–24, 2004.
[10] Hirschfeld R. AspectS aspect-oriented programming with squeak. In: Aksit M, Mezini M, Unland R, editors. Objects components architectures services and applications for a networked world, Lecture Notes in Computer Science, vol. 2591. Berlin: Springer; 2003. p. 216–32.
[11] Masuhara H, Kiczales G, Dutchyn C. A compilation and optimization model for aspect-oriented programs. In: Proceedings of compiler construction (CC), Lecture Notes in Computer Science, vol. 2622, Berlin: Springer, 2003. p. 46–60.
[12] Hanenberg S, Hirschfeld R, Unland R. Morphing aspects: incompletely woven aspects and continuous weaving. In: Proceedings of the conference on aspect-oriented software development (AOSD), March 22–26, Lancaster, UK: ACM Press, 2004. p. 46–55.
[13] Ingalls D, Kaehler T, Maloney J, Wallace S, Kay A. Back to the future: the story of squeak, a practical smalltalk written in itself. In: Proceedings of the conference on object-oriented programming systems, languages and applications (OOPSLA), October 5–9, Atlanta, GA, USA: ACM Press, 1997. p. 318–26.
[14] ChiMu Corporation. Java and Smalltalk syntax compared. (⟨http://www.chimu.com/publications/JavaSmalltalkSyntax.html⟩) 2000.
[15] Goldberg A, Robson D. Smalltalk 80—the language and its implementation. Reading, MA: Addison-Wesley; 1983.
[16] Brant J, Foote B, Johnson R.E., Roberts D. Wrappers to the rescue. In: Proceedings of the European conference on object-oriented programming (ECOOP), Lecture Notes in Computer Science, vol. 1445, Springer, 1998. p. 396–417.
[17] Kiczales G, des Rivieres J, Bobrow DG. The art of the metaobject protocol. Reading, MA: Addison-Wesley; 1991.
[18] Kniesel G, Costanza P, Austermann M. JMangler—a powerful back-end for aspect-oriented programming. In: Filman R, Elrad T, Clarke D, Aksit M, editors. Aspect-oriented software development. Englewood Cliff, NJ: Prentice Hall; 2004.
[19] Chiba S. Load-time Structural Reflection in Java. In: Proceedings of the European conference on object-oriented programming (ECOOP), Lecture Notes in Computer Science, vol. 1850. Berling: Springer, 2000. p. 313–36.
[20] Bretthauer H. Kopp J. Balancing the EuLisp metaobject protocol. In: Proceedings of the international workshop on new models for software architecture, Tokyo, Japan, 1992.
[21] Sato Y, Chiba S, Tatsubori M. A selective, just-in-time aspect weaver, In: Proceedings of the conference on generative programming and component engineering (GPCE), Lecture Notes in Computer Science, vol. 2830. Berlin: Springer, 2003. p. 189–208.
[22] Gabriel R, White J, Bobrow D. CLOS: Integrating object-oriented and functional programming. Communications of the ACM 1991;34(9): 28–38.
[23] de Alwis B. Aspects of incremental programming. Master's Thesis, University of British Columbia, Vancouver, Canada, 2002.
[24] de Alwis B, Kiczales G. Apostle: a simple incremental weaver for a dynamic aspect language. Technical Report TR-2003-16, Department of Computer Science, University of British Columbia, Vancouver, Canada, 2003.
[25] Mezini M. Maintaining the consistency of class libraries during their evolution. In: Proceedings of the conference on object-oriented programming systems, languages and applications (OOPSLA), October 5–9, Atlanta, GA, USA: ACM Press, 1997, p. 1–22.