Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts

Andy Kellens[1⋆], Kim Mens[2], Johan Brichau[1,3], and Kris Gybels[1]

[1] Programming Technology Lab
Vrije Universiteit Brussel, Belgium
{ akellens | jbrichau | kris.gybels }@vub.ac.be
[2] Département d'Ingénierie Informatique
Université catholique de Louvain, Belgium
kim.mens@uclouvain.be
[3] Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille, France

Abstract. In spite of the more advanced modularisation mechanisms, aspect-oriented programs still suffer from evolution problems. Due to the *fragile pointcut problem*, seemingly safe modifications to the base code of an aspect-oriented program can have an unexpected impact on the semantics of the pointcuts defined in that program. This can lead to broken aspect functionality due to accidental join point misses and unintended join point captures. We tackle this problem by declaring pointcuts in terms of a conceptual model of the base program, rather than defining them directly in terms of how the base program is structured. As such, we achieve an effective decoupling of the pointcuts from the base program's structure. In addition, the conceptual model provides a means to verify where and why potential fragile pointcut conflicts occur, by imposing structural and semantic constraints on the conceptual model, that can be verified when the base program evolves. To validate our approach we implemented a *model-based pointcut* mechanism, which we used to define some aspects on SmallWiki, a medium-sized application, and subsequently detected and resolved occurrences of the fragile pointcut problem when this application evolved.

1 Introduction

Ever since its inception almost ten years ago, aspect-oriented software development (AOSD) has been promoted as a powerful development technique that extends the modularisation capabilities of existing programming paradigms such as object orientation [1]. To this extent, aspect-oriented programming languages provide a new kind of modules, called *aspects*, that allow one to modularise the implementation of crosscutting concerns which would otherwise be spread across various modules. The resulting improved modularity and separation of concerns

⋆ Ph.D. scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

intends not only to aid initial development, but also to allow developers to better manage software complexity, evolution and reuse [2]. Given the fact that maintenance and evolution of software applications account for the largest part of the software development process [3], the introduction and use of AOSD techniques looks promising.

Paradoxically, the essential techniques that AOSD proposes to improve software modularity seem to restrict the evolvability of that software. AOSD puts forward that aspects are not *explicitly* invoked but instead, are *implicitly* invoked [4]. This has also been referred to as the 'obliviousness' property of aspect orientation [5]. It means that the developer of the *base program* (i.e., the program without the aspects) does not need to explicitly invoke the aspects because the aspects themselves specify when and where they need to be invoked, by means of a *pointcut definition*. As a consequence, these pointcut definitions typically rely heavily on the structure of the base program.

This tight coupling of the pointcut definitions to the base program's structure and behaviour can hamper the evolvability of the software [6]: it implies that all pointcuts of each aspect need to be checked and possibly revised whenever the base program evolves. Indeed, due to changes to the base program, the pointcuts may unanticipatedly capture join points that were not supposed to be captured, or may no longer capture join points that should have been affected by the aspect. This problem has been coined *the fragile pointcut problem* [7, 8].

We address the fragile pointcut problem by replacing the intimate dependency of pointcut definitions on the base program by a more stable dependency on a conceptual model of the program. These *model-based pointcut* definitions are less likely to break upon evolution, because they are no longer defined in terms of how the program happens to be structured at a certain point in time.

Because model-based pointcut definitions are decoupled from the actual structure of the base program, the fragile pointcut problem is thus transferred to a more conceptual level. Whereas traditional pointcut definitions may cause unanticipated captures and accidental misses of program entities upon evolution of the base program, model-based pointcut entities may lead to mismatches between the conceptual model of the program and the program entities to which the model is mapped. Hence, the fragile pointcut problem is transformed into the problem of keeping a conceptual model of the program synchronised with that program, when the program evolves.

To solve this derived problem, we rely on previous research that enables documenting the program structure and behaviour at a more conceptual level, where appropriate support is provided for keeping the 'conceptual model documentation' consistent with the source code when the program evolves. More specifically, we implement our particular solution to the fragile pointcut problem through an extension of the CARMA aspect language [9] combined with the formalism of *intensional views* [10]. The resulting approach tightly integrates the intensional views development tool with an aspect-oriented language. In essence, the pointcuts defined in the aspect language rely on the model that is built using the development tool. We validate our solution on SmallWiki, a medium-sized

Smalltalk application, where we illustrate how fragile pointcuts are detected and resolved more easily using model-based pointcuts, as opposed to using more traditional pointcuts.

2 The Fragile Pointcut Problem

In this section, we define the *fragile pointcut problem*, provide an analysis of possible causes of fragility of pointcut definitions, and illustrate each of them through a running example. We then study the fundamental causes underlying the problem, which will lead to our solution of *model-based pointcuts*.

2.1 Definitions

According to Stoerzer et al. [7, 8], pointcuts are *fragile* because their semantics may change 'silently' when changes are made to the base program, even though the pointcut definition itself remains unaltered. The semantics of a pointcut change if the set of join points that is captured by that pointcut changes. Several other authors have observed symptoms of the fragile pointcut problem [6, 11]. Before elaborating on these observations, we define the fragile pointcut problem:

> The **fragile pointcut problem** occurs in aspect-oriented systems when pointcuts unintentionally capture or miss particular join points as a consequence of their fragility with respect to seemingly safe modifications to the base program.

Therefore, in an aspect-oriented program, one cannot tell whether a change to the base code is safe simply by examining the base program in isolation. All pointcuts referring to the base program need to be examined as well.

Intuitively, because pointcuts capture a set of join points based on some structural or behavioural property shared by those join points, any change to the structure or behaviour of the base program can impact the set of join points that is captured by the pointcut definitions. If, upon evolution of the base program, source-code entities are altered which *accidentally* leads to the capture of a join point related to these source-code entities, we say that we have an **unintended join point capture**. Conversely, when the base program is changed in such a way that one of the join points that was originally captured by the pointcut is no longer captured, even though it was still supposed to be captured, we say we have an **accidental join point miss**. We define the **join point mismatches** (w.r.t. a given pointcut) as the union of the unintended join point captures and the accidental join point misses.

In literature, we find some interesting observations that confirm the existence of the fragile pointcut problem. Kiczales and Mezini [11] identified that aspects 'cut new interfaces' through the modules of a system and state that, in the presence of aspects, the complete interface of such a module can only be determined once the complete configuration of all modules in the system is known. Sullivan et al. [6] observed that the criterion of *obliviousness* in AOSD comes

at a considerable cost to aspect designers. They describe how aspect designers are confronted with complex pointcut definitions and extreme sensitivity of the validity of pointcuts to changes in the base program.

The fragile pointcut problem can be considered as the aspect-oriented equivalent of the *fragile base class problem* [12] found in object-oriented software development. In the fragile base class problem, one cannot tell whether a change to a base class is safe simply by examining the base class' methods in isolation; instead, one should examine all subclasses of that base class as well [13]. Analogously, in the fragile pointcut problem, one cannot tell whether a change to any part of the base program is safe without examining all pointcut definitions and determining the impact of that local change on each pointcut definition.

2.2 Examples

To understand the fundamental causes underlying the fragile pointcut problem, we study its various instantiations, and analyse how different kinds of pointcut definitions are fragile with respect to evolution of the base program. We observe that fragility of a pointcut depends on three fundamental properties of a pointcut definition:

1. The technique used to define the pointcut (e.g., enumeration of join points, pattern-based matching, ...);
2. The expressiveness of the pointcut language (i.e., the structural and behavioural properties available to capture join points);
3. The join point model, more particularly, the kinds of join points that can be captured by a pointcut (method executions, method calls, variable assignments, ...)

We illustrate the impact of these properties on the fragility of pointcuts, using a simple example: the Java implementation of a buffer object with a synchronisation aspect:

```
class Buffer {
       private Object content[];
       private int index = 0;
       ...
       public Object get() {
              ... return content[index] ... };
       public void set(Object el) {
              ... content[index] := el ... };
       ...
 }
```

The implementation of a synchronisation aspect for this buffer contains a pointcut that captures all calls to the `get()` and `set()` accessor methods. Depending on the technique used to define it, the pointcut is fragile w.r.t. different modifications of the base program.

Enumeration pointcut The simplest definition for this 'accessors' pointcut merely enumerates all join points that need to captured, by their exact signature:

```
pointcut accessors()
       call(void Buffer.set(Object)) || call(Object Buffer.get());
```

This pointcut definition is particularly fragile to accidental join point misses. Any change to the signature of the accessor methods requires a revision of the pointcut definition. Furthermore, consider an evolution of the buffer implementation where additional accessors are defined: e.g., the addition of `setAll` and `getAll` methods that get or set multiple objects at once in the buffer. Such an evolution requires revising the pointcut definition to explicitly add all new accessor methods to it. Otherwise, the pointcut would exhibit accidental misses of the call join points to these new accessor methods, and the synchronisation aspect would fail.

Pattern-based pointcut In a pattern-based pointcut, we capture the desired join points by specifying a pattern, for example using wildcards over the signature. The following pattern captures all calls to methods of which the name starts with `set` or `get`:

```
pointcut accessors()
       call(* set*(..) ) || call(* get*(..) );
```

This pointcut is also fragile w.r.t. evolution of the base program. New methods can be added and existing ones can be removed such that they are captured by the pointcut definition, as long as they follow the naming convention encoded in the pattern. In addition, consider an evolution of the base code where a method named `setting` is added. A call to this method is unintendedly captured by the pointcut because its name happens to start with `set`.

Structural property-based pointcuts In more advanced pointcut languages that allow to extract fine-grained structural properties of program elements to describe the join points, we can declare accessor methods as those methods that either assign to or return an instance variable directly. The following pointcut uses an AspectJ-like syntax[4] to illustrate a property-based pointcut that can, for example, be expressed in the CARMA pointcut language [9]. In CARMA, variables in pointcut definitions are prefixed with `?`. The first pointcut expression captures all calls to methods that assign to an instance variable and the second pointcut expression captures all calls to 'getter' methods. The `assigns` and `returnsVariable` predicates reify the structural property of which variables that are assigned to or returned by the method[5]. The `instanceVariable` predicate reifies the instance variables defined in a class.

[4] We use this hypothetical AspectJ-like syntax to avoid having to explain the details of the CARMA syntax here.

[5] The predicates also consider indexing in arrays for variable accesses and assignments.

```
pointcut setters()
       call(?class.?method(..) ) &&
       assigns(?class.?method,?iv)  &&
       instanceVariable(?iv,?class);
pointcut getters()
       call(?class.?method(..) ) &&
       returnsVariable(?class.?method,?iv) &&
       instanceVariable(?iv,?class);
```

Although these pointcuts are no longer fragile w.r.t. changes in the name of the methods, they are still fragile because they capture only methods that respect the structural convention codified by the pointcut. Consider, for example, the following 'getter' method that does not directly return the instance variable in a return statement but returns another (temporary) variable:

```
Object get() {
   Object temp := content[index];
   ..
   return temp;
}
```

Although the variable `temp` contains the actual value of the instance variable, a call join point to this method would be missed by our previous pointcut definition. Hence, once again, the pointcut is fragile to changes in the base program's source code.

Behavioural property-based pointcuts Behavioural properties that can be used in pointcut definitions mostly concern an application's execution history or runtime values during that history. A well-known behavioural property to qualify pointcuts is determined by the `cflow` predicate. Using `cflow`, we can specify join points that lie in the control flow of other join points. For example, the following pointcut captures only those join points that are 'getter' join points (as defined previously) but do not lie in the flow of control of other 'getter' join points. Using this `optimisedGetter` pointcut, we can prevent the execution of the synchronisation aspect if the running thread is already in the control flow of the synchronisation aspect (i.e., if the buffer is already synchronised).

```
pointcut optimisedGetter() :
       getters() &&
       !cflow(getters());
```

However, even behavioural property-based pointcuts are fragile to evolution of the source code, because they also need to refer to the source-code entities of which they want to characterise the behaviour. In this particular example, the pointcut is defined in terms of the `getters` pointcut. Because that latter pointcut is fragile, the `optimisedGetter` pointcut is equally fragile. Mind that this fragility also holds for many pointcuts that use dynamic values (of e.g. instance variables) because they often need to refer to the actual instance variables, of which they use the values, by name.

Uncapturable join points While the previous examples illustrated the fragility of the pointcut due to the definition technique or the provided expressiveness of the pointcut language, another major reason for fragility lies in the fact that some intended join points simply cannot be captured because:

- The join point model is too restrictive and the code to be advised by the aspect is not confined to a join point. For example, most aspect languages today do not allow to advice pieces of method bodies. In our buffer example, this would mean that we must structure the possible critical sections in the buffer implementation as complete methods. Otherwise, they cannot be advised by the synchronisation aspect.
- The pointcut cannot be described because the join points do not share sufficient structural or behavioural properties to allow them to be qualified in a pointcut definition. As a consequence, developers are forced to use fragile enumeration-based pointcuts.

2.3 Problem Analysis

In all of the examples above, pointcuts are fragile because their definitions are tightly coupled to a particular structure or behaviour of the base program. Similar to how most programming paradigms rely on symbolic referencing (e.g. function calls by name), aspect-orientation relies on referencing more intricate structural and behavioural properties of the program as well. More precisely, pointcuts impose 'design rules' that developers of the base program must adhere to in order to prevent unintended join point captures or accidental join point misses (also see [6]). These rules originate from the fact that pointcuts try to define intended conceptual properties about the base program, based on structural and behavioural properties of the program. For example, the 'accessors' pointcut tries to define the conceptual property of an 'accessor method' by relying on coding conventions used to implement that method. Therefore, in general, base program developers need to adhere to such rules when implementing the base program, so that the pointcut definition can be expressed in terms of those rules. Because the rules themselves are not enforced by any mechanism, not only do the developers need to be aware of these rules, they also need to manually ensure not to break them when evolving the base program. This requires very disciplined developers that have a good understanding of the actual rules that the pointcut definitions depend on. Consequently, in practice these rules are likely to be violated upon evolution.

While the design rules imposed by enumeration-based pointcuts are very restrictive (i.e., only the explicitly enumerated join points can be advised), behavioural property-based pointcuts allow for more (structural) diversity in the base program but are also more complex to understand, write and verify. For example, we could define the 'accessors' pointcut by relying on the behavioural property that the method returns an instance variable value. However, this behavioural property cannot be statically verified upon program evolution in all cases. Moreover, although there are 'behavioural' design rules (that can be expressed using advanced pointcut languages [9, 14, 15]) that do not need to refer

to structural properties in the program's source code, such structural properties are still required in many cases.

To the best of our knowledge, none of the proposed solutions that exist today (pointcut delta analysis [7], expressive pointcut languages [9, 14, 15], source-code annotations [16, 17], design rules [6]) address *both* the too tight coupling of pointcuts to the structure of the program, and the brittleness of the imposed design rules upon program evolution. In the next section, we introduce a novel technique to define pointcuts, that achieves low coupling and provides a means to detect violations of the imposed rules. This technique is orthogonal to the techniques mentioned above, which are described in section 7.

3 Model-based Pointcuts

We tackle the fragile pointcut problem with *model-based pointcuts*. This new pointcut definition mechanism achieves a low coupling of the pointcut definition with the source code, while at the same time providing a means of documenting and verifying the design rules on which the pointcut definitions rely.

Model-based pointcut definitions are defined in terms of a conceptual model of the base program, rather than referring directly to the implementation structure of that base program. Figure 1 illustrates this difference between *model-based* and traditional *source-code based* pointcuts. On the left-hand side, a traditional source-code based pointcut is defined directly in terms of the source code structure. On the right-hand side, a model-based pointcut is defined in terms of a conceptual model of the base program. This conceptual model provides an abstraction over the structure of the source code and classifies base program entities according to the concepts that they implement. As a result, model-based pointcuts capture join points based on conceptual properties instead of structural properties of the base program entities. In addition to decoupling the pointcut definitions from the base program's implementation structure, the classifications in the conceptual model are specifically conceived to provide support for detecting evolution conflicts between the conceptual model and the base program.

For example, assuming that the conceptual model contains a classification of all accessor methods in the buffer implementation, the model-based pointcut that captures all call join points to these accessor methods could be defined as:

```
pointcut accessors():
     classifiedAs(?methSignature,AccessorMethods) &&
     call(?methSignature);
```

where the expression `classifiedAs(?methSignature,AccessorMethods)` matches all methods that are classified as accessor methods in the conceptual model of the buffer program and the variable `?methSignature` is bound to the method signature of such a method. This pointcut definition explicitly refers to the concept of an accessor method rather than trying to capture that concept by relying on implicit rules about the base program's implementation structure. Consequently, this pointcut does not need to be verified or changed upon evolution of

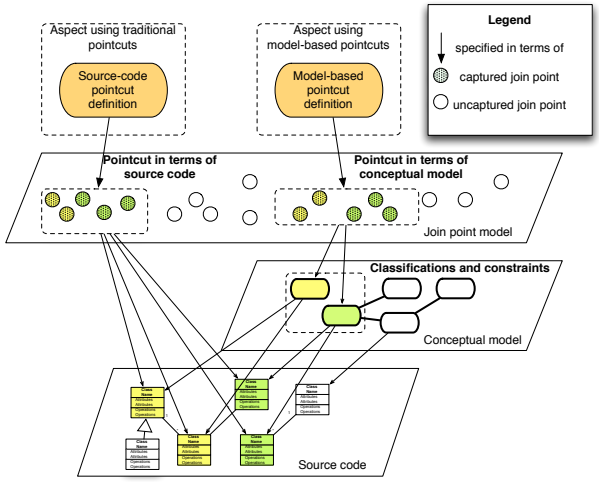

Fig. 1. Traditional pointcuts versus model-based pointcuts

the base program: if the conceptual model correctly classifies all accessor methods, this pointcut remains correct. In a certain sense, model-based pointcuts are similar to Kiczales and Mezini's *annotation-call* and *annotation-property* pointcuts [17]. Indeed, the classifications of source-code entities in the conceptual model could be constructed using annotations in the source code.

By defining pointcuts in terms of a conceptual model, the fragile pointcut problem has now been translated into the problem of keeping the classifications of the conceptual model consistent with the base program. To detect incorrectly classified source entities, the conceptual model goes beyond mere classification or annotation and defines extra constraints over the classifications that need to be respected by the source-code entities, for the model to be consistent. Formally, we distinguish two cases, defined below and illustrated by figure 2:

1. We define the set of possible *unintended captures* for a concept A as those entities that are classified as belonging to A but that do not satisfy some of the constraints defined on A:

$$UnintendedCaptures_A = \bigcup_{C \in \mathcal{C}_A} (A - ext(C))$$

   where $\mathcal{C}_A$ is defined as the set of all constraints on A and $ext(C)$ denotes the set of all source-code entities satisfying constraint C. The intuition behind this definition is that if an entity belongs to A but does not satisfy the constraints defined on A then maybe the entity is misclassified.
2. We define the set of possible *accidental misses* as those entities that do not belong to A, but do satisfy at least one of the constraints C defined on A:

$$AccidentalMisses_A = \bigcup_{C \in \mathcal{C}_A} (ext(C) - A)$$

The intuition behind this definition is that if an entity does not belong to A but does satisfy some of the constraints defined on A, then maybe the entity should have been classified as belonging to A. To avoid having an overly restrictive definition (yet at the risk of having a too liberal one), we do not require the missed entity to satisfy *all* constraints defined on A. As soon as it satisfies one constraint, we flag it as a *potential* accidental miss.

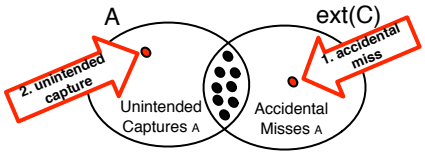

Fig. 2. Detecting potential unintended captures and accidental misses

The set of all *potential* unintended captures and accidental misses that can be detected is then defined as

$$Mismatches_A = \bigcup_{C \in \mathcal{C}_A} (A \; \Delta \; ext(C))$$

where Δ denotes symmetric difference. Whenever there is an unintended capture (resp. accidental miss) this can have one of 3 possible causes :

1. Either a source-code entity was misclassified and should be removed from (resp. added to) A;
2. Either a constraint C no longer applies and thus needs to be modified or removed;
3. Either a source-code entity accidentally satisfies (resp. invalidates) a constraint C and should be adapted.

In summary, model-based pointcuts provide support for detecting and resolving occurrences of the fragile pointcut problem because:

- Model-based pointcut definitions are decoupled from the source-code structure of the base program. They explicitly refer to a conceptual model of the program that classifies base program entities according to concepts that are of interest to define pointcuts.
- Although the conceptual model still classifies base program entities based on their implementation structure, the model does include constraints that allow verification of the consistency of the program's source code with respect to the classifications, when the source code evolves.

In practice, model-based pointcuts offer aspect developers a means to extract the structural dependencies from the pointcut definition and move these dependencies to the conceptual model specification, where they can be more easily enforced and checked. Upon evolution of the base program, the 'design rules' that govern these structural dependencies are automatically verified and the developer is notified of possible conflicts of the source code w.r.t. those rules.

4 View-based Pointcuts

As a particular instantiation of model-based pointcuts, we introduce *view-based pointcuts*, which:

1. use the formalism of *intensional views* [10], and its associated tool suite IntensiVE[6], to express a conceptual model of a program and to keep it synchronised with the source code of that program;
2. specify pointcuts in terms of this conceptual model, using an extension to the aspect-oriented language *CARMA*.

We briefly present the formalism of intensional views and how it can be used to define a conceptual model of a program. Next, we introduce the CARMA aspect language and its extension to define aspects over intensional views. Throughout this section we use examples taken from the SmallWiki system. Section 5 explains this case in more detail.

4.1 Intensional Views

In earlier work [10], we presented the formalism of intensional views as a technique for describing a conceptual model of a program's structure and verifying consistency of that model with respect to the program. For the sake of completeness, we briefly repeat the formalism here, with a particular focus on those features that enable it to detect interesting evolution conflicts.

Intensional views describe concepts of interest to a programmer by grouping program entities (classes, methods, ...) that share some structural property. These sets of program entities are specified intensionally, using the logic metaprogramming language *Soul* [18]. (The *intension* — with an 's' — of a set is its description or defining properties, i.e., what is true about the members of the set. The extension of a set is its members or contents.)

For example, to model the concept of "all actions on Wiki pages" (save, login,...) in SmallWiki, we specify an intensional view named *Wiki Actions*, which groups all methods of which the name starts with `execute`, based on the observation that all action methods indeed respect that naming convention:

```
classInNamespace(?class,[SmallWiki]),
methodWithNameInClass(?entity,?name,?class),
['execute*' match: ?name asString]
```

Without explaining all details of the Soul syntax and semantics, upon evaluation the above query accumulates all solutions for the logic variable `?entity`, such that `?entity` is a method, implemented by a class in the `SmallWiki` namespace, whose name starts with `execute`. This query is the *intension* of the view.

Since the declared intension is sometimes too broad or too restrictive with respect to the actual program code, intensional views provide means to deal with *deviations* to a view, allowing to explicitly 'include' or 'exclude' specific

[6] Available for download on http://www.intensional.be

program entities from a view. For example, if the SmallWiki implementation would contain a method that starts with `execute` but does not perform an action, we would put that method in the *excludes set* of the *Wiki Actions* view.

Upon evolution of the program, a simple view such as the one defined above can capture or miss particular program entities accidentally, which is similar to the fragile pointcut problem. Therefore, a set of constraints on and between views (as defined in Section 3) is at the heart of the intensional views formalism. This set of constraints can be validated with respect to the program code and allows keeping an intensional view model synchronized with the program. We highlight two different types of constraints that can be defined on intensional views: *alternative intensions* and *intensional relations*.

Alternative Intensions. Often, the same set of program entities can be specified in different ways, e.g. when they share multiple naming or coding conventions. A first kind of constraints that can be declared on an intensional view is through the definition of multiple alternative intensions for that view. Each of these alternatives is required to be *extensionally consistent*, meaning that they need to describe exactly the same set of program entities. For example, all methods performing actions on Wiki pages do not only have a name that starts with `execute`, but are also implemented in a method protocol[7] called `action`. We can therefore define the *Wiki Actions* view in an alternative way, using a logic query that accumulates all SmallWiki methods implemented in that protocol. Since both alternatives are supposed to define the same concept (i.e. Wiki actions), we require both alternatives to capture the same set of methods.

Intensional Relations. Whereas alternative intensions declare an equality relation between the different alternatives of a view, a second means of specifying constraints is through *intensional relations*, which are binary relations between intensional views, of the canonical form:

$$\mathcal{Q}_1\ x \in View_1 : \mathcal{Q}_2\ y \in View_2 : x\ R\ y$$

where $\mathcal{Q}_i$ are logic quantifiers ($\forall$, $\exists$, $\exists!$ or $\nexists$), $View_i$ are intensional views, and R is a verifiable binary relation over the source-code entities (denoted by x and y) contained in those views. An example of such a constraint in SmallWiki is that all *Wiki Actions* should be implemented by *Action Handlers*. Assuming we have defined an *Action Handlers* view (a set of classes implementing actions), we express this dependency as:

$$\forall\ x \in WikiActions : \exists\ y \in ActionHandlers : x\ \texttt{isImplementedBy}\ y$$

where `isImplementedBy` is a binary predicate which verifies that a given method x is implemented by a given class y. Like for intensional views, explicit *deviations* can be declared on intensional relations.

[7] In Smalltalk, the methods of a class are organised in logical groups called protocols.

As for checking extensional consistency, the IntensiVE tool suite can be used to verify the validity of the constraints imposed by intensional relations with respect to the program code. As explained in Section 3, invalidations of these constraints either indicate unintended captures or accidental misses, or maybe the constraint itself is simply no longer valid and should be modified or removed.

4.2 View-based Pointcuts in CARMA

Having chosen the formalism in which to express the conceptual model of the base program, we still need a pointcut language that permits us to define pointcuts in terms of that model. Given that both the formalism of intensional views and the aspect-oriented programming language CARMA rely on the logic metaprogramming language Soul, to specify intensions and pointcuts respectively, we extended CARMA with the ability to define *view-based pointcuts*.

- reception(?joinpoint, ?message, ?arguments)
  Expresses that **?joinpoint** is a *message reception join point*, where the message with name **?message** is received with the arguments in the list **?arguments**.
- send(?joinpoint, ?message, ?arguments)
  The join point **?joinpoint** is a *message send join point* where the message with name **?message** is sent and passed the arguments in the list **?arguments**.
- within(?joinpoint, ?class, ?method)
  The join point **?joinpoint** is lexically associated to the method named **?method** in class **?class**. This means the join point was raised because of an expression in the body of that method or because of the execution of that method itself.
- withinClass(?joinpoint, ?class)
  The join point **?joinpoint** is lexically associated to the class **?class**. This means the join point was raised because of an execution of a method defined on that class or because of an expression in the body of a method of that class.

Fig. 3. Some basic predicates provided by CARMA for capturing join points

CARMA is very similar to the AspectJ language but features a logic pointcut language, and is an aspect-oriented extension to Smalltalk instead of Java. Pointcuts in CARMA are logic queries that can express structural as well as dynamic conditions over the join points that need to be captured by the pointcut. To this extent, a query can make use of a number of predefined predicates, stating conditions over join points, which form the heart of the CARMA language. Some basic CARMA predicates that are used in this paper are shown in Figure 3.

The expressive power of CARMA is a direct consequence of the logic language features of unification and recursive logic rules, together with a complete and open reification of the entire base program. CARMA has already proven useful to write more robust property-based and pattern-based pointcut definitions [9]. For this paper, we further enhanced CARMA with view-based pointcuts using an additional predicate `classifiedAs(?entity,?view)` that allows to define join

points in terms of the intensional views defined over a program. For example, we can define a view-based pointcut that captures all calls to methods contained in the `Wiki Actions` view as:

```
pointcut wikiActionCalls():
     classifiedAs(?method,WikiActions),
     methodInClass(?method,?selector,?class),
     send(?joinpoint,?selector,?arguments)
```

The above pointcut definition is tightly coupled to the intensional view model of SmallWiki but it is decoupled from the actual program structure. In combination with the support for verifying consistency of the intensional views model with respect to the source code, we can thus alleviate part of the fragile pointcut problem, as is illustrated by the experiment in the following section.

5 Experiment: Aspects in SmallWiki

In this section, we demonstrate on a small but realistic program, on which two aspects were defined, how view-based pointcuts can detect occurrences of the fragile pointcut problem when the program evolves.

Case selection. The case study we selected is SmallWiki [19], a fully object-oriented and extensible Wiki framework, written by Lukas Renggli, that was developed entirely in VisualWorks Smalltalk. A Wiki is a collaborative web application that allows users to add content, but also allows anyone to edit the content. The original version of SmallWiki we studied was version 1.54, the first internal release of SmallWiki (14-12-2002), offering an operational Wiki server with rather limited functionality: only the rendering and editing of fairly simple Wiki pages was supported. This version contained 63 classes and 424 methods.

Set-up of the experiment. To illustrate our approach we conducted the following experiment:

1. We identified two aspects to be defined on the SmallWiki case.
2. Using an AOP approach with *traditional* pointcuts, we extended version 1.54 of SmallWiki with the extra functionality described by the aspects.
3. We applied the same aspects to version 1.304 of SmallWiki, an *evolved* version of the application dating one year after the release of version 1.54. We analyzed the impact of the changes in that evolution on the aspects. In particular, we assessed which changes gave rise to the *fragile pointcut problem*.
4. We made a *conceptual model* of Smallwiki's program structure. In practice, we merely reused a conceptual model which we conceived for an earlier experiment using intensional views and relations. (By selecting a set of intensional views that were already defined on the application, even before the aspects were identified, we show that our approach does not necessarily require the views to be defined in function of the aspects.)

5. We implemented the two aspects by means of *view-based pointcuts*, defined in terms of that *conceptual model*.
6. We reapplied these aspects to the evolved version of SmallWiki and observed how the evolved program gave rise to conflicts between the conceptual model and the program. We compared the conflicting program entities with those that caused the fragile pointcut problem before. By using the feedback of the IntensiVE tool suite, we brought the program in sync with the conceptual model, and analysed the implication of these changes in the light of the fragile pointcut problem.

In the next subsections we elaborate on each of the steps of our experiment.

5.1 Two Aspects in SmallWiki

We extended our initial version 1.54 of SmallWiki with two simple aspects:

1. *logging of actions*: this aspect outputs information concerning which actions (like saving, opening a page, ...) are executed in SmallWiki.
2. *output in italics*: this aspect changes the output of SmallWiki by rendering all text in italics instead of a regular font.

5.2 A Traditional AOP Implementation

We implemented these two aspects in the (non-extended version of the) *CARMA* aspect language. Below, we highlight how we defined the pointcuts in terms of which those aspects were defined.

Implementing the 'logging' aspect. As mentioned earlier, all actions in the Wiki system are implemented by means of a method which starts with the string `execute`. Using this information we write down the following pointcut for the *logging of actions* aspect:

```
1   classInNamespace(?class,[SmallWiki]),
2   methodWithNameInClass(?method,?selector,?class),
3   ['execute*' match: ?selector],
4   reception(?joinpoint,?selector,?arguments)
```

Line 1 of this pointcut selects all classes in the `SmallWiki` namespace; line 2 and 3 select all methods within those classes whose name start with the string `execute`; finally, line 4 selects all message reception join points of those methods.

Implementing the 'output in italics' aspect. The output of Wiki pages is rendered by visitor objects which, for each different page component, generate some kind of output (HTML, Latex, save to disk). We wish to weave on all calls to these visitors originating from a Wiki page element. We declare this by means of the following pointcut:

```
1 classInNamespace(?class,[SmallWiki]),
2 or( classInHierarchyOf(?class,[PageComponent]),
3     classInHierarchyOf(?class,[Structure])),
4 classInHierarchyOf(?outputclass,[OutputVisitor]),
5 methodWithNameInClass(?method,?name,?outputclass),
6 within(?joinpoint,?class,?m),
7 send(?joinpoint,?name,?args)
```

Lines 1–3 collect all page element classes (i.e., all classes in SmallWiki which are either located in the hierarchy of `PageComponent` or in the hierarchy of `Structure`); lines 4–5 select all methods that render output (i.e., methods implemented on classes in the `OutputVisitor` hierarchy); finally, lines 6 and 7 select all join points from within Wiki page elements which perform a message send to a method that renders output.

5.3 Applying the aspects to the evolved application

The evolved version of SmallWiki we selected was version 1.304, an internal release of almost a full year (16-11-2003) after the 1.54 release (14-12-2002). With 108 classes and 1219 methods, this evolved version was significantly larger than version 1.54.

When assessing the impact of this evolution on the pointcut of the *logging* aspect, we observed that all but two actions were correctly captured by the pointcut. The `save` and `authenticate` actions, which were added in version 1.304, were *not* captured by the pointcut, because their method names do not start with the string `execute`. The addition of these two methods thus caused two accidental join point misses in the evolved version of SmallWiki.

We mentioned earlier that the 'Wiki actions' are not only characterized by the fact that they all start with the string `execute`, but that they are also all categorized in a method protocol named `action`. We could have expressed the pointcut in terms of this alternative coding convention. This however would also have resulted in a join point mismatch when applying the pointcut to the evolved version of SmallWiki. Two other execute methods, namely `executeSearch` and `executePermission`, would have been missed by the pointcut because they have been misclassified in the `private` protocol instead of the `action` protocol.

The evolution step also had an impact on the pointcut for the *italic output* aspect. In the evolved version, a significant number of new Wiki page element components were added. For a number of these (i.e., `LinkInternal` and `LinkMailTo`) there did not exist a directly corresponding visit method (i.e., `visitLinkInternal` and `visitLinkMailTo`) in the `OutputVisitor` hierarchy, as was implicitly assumed by the pointcut declaration. Instead, for reasons of implementation reuse, they had their visit method implemented on the abstract `Visitor` class. From within this more abstract visit method, other visit methods, which *were* part of the `OutputVisitor` hierarchy, were then called. This subtle change in the implementation had a major impact on the correctness of the pointcut, causing the output generation methods of some of the newly added classes to be accidentally missed by the pointcut.

5.4 Intensional Views on SmallWiki

In a previous experiment we documented the conceptual structure of SmallWiki with 17 intensional views and 16 intensional relations [10]. For the current experiment, we reused that conceptual model, modulo the renaming of some intensional views to better reflect what SmallWiki concepts they represent. We do not show all views here[8], but limit ourselves to those interesting for the remainder of this paper, as summarized by Figure 4, and explained below.

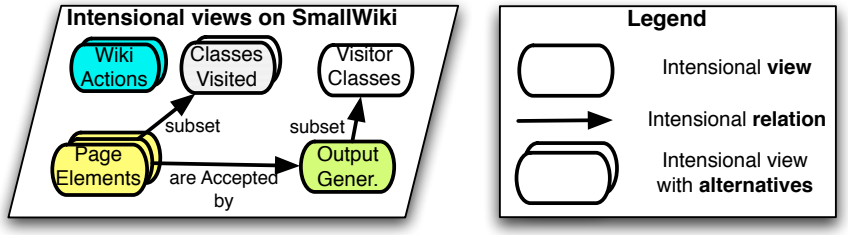

Fig. 4. Part of the conceptual model of SmallWiki version 1.54

- The *Wiki Actions* view groups all methods implementing an action on a Wiki page. Two alternative intensions for this view were explained in subsection 4.1.
- The *Page Elements* view groups all classes representing components out of which Wiki pages can be constructed (e.g., text, links, tables, lists). This view is defined by the following *alternative* intensions:
  1. All subclasses of either the `PageComponent` or `Structure` class;
  2. All classes in the Wiki system implementing a method named `accept`;
  3. All classes in the Wiki system containing a protocol named `visiting`.

  Alternatives 2 and 3 codify the knowledge that, in order to implement operations on Page Elements, SmallWiki relies heavily on the Visitor pattern.
- The *Output Generation* view groups all classes that generate output (e.g., HTML, Latex) for (the components of) a Wiki Page, and has as intension all subclasses of `OutputVisitor`.
- The *Visitor Classes* view groups all subclasses of `Visitor`.
- The *Classes Visited* view describes another aspect of the Visitor pattern and groups all classes that are visited by some `Visitor` in the SmallWiki system.

In addition to the constraint of extensional consistency between the different alternatives of each of these views, Figure 4 also shows 3 intensional relations that impose additional constraints on the views:

[8] For a more exhaustive list of views that were defined on SmallWiki, and how they were defined, see [10].

– The relation that the *Page Elements* view is a subset of the *Classes Visited* view codifies the knowledge that all page elements can be visited by a vistor.
– The `are Accepted by` relation captures the important fact that all page elements can be rendered by an *Output Generation* visitor.
– The third intensional relation states that the classes that render output are a particular kind of *Visitor Classes*.

5.5 Implementation with View-Based Pointcuts

In this subsection we show how we defined the pointcuts of the two aspects introduced earlier as *view-based pointcuts* in terms of the views discussed in subsection 5.4.

To implement the *logging of actions* aspect using a view-based pointcut, we use the following pointcut definition:

```
1 classifiedAs(?method,WikiActions),
2 methodWithName(?method,?message),
3 reception(?joinpoint,?message,?args),
4 withinClass(?joinpoint,?class)
```

This pointcut selects all reception join points of a message which is implemented by a method in the *Wiki Actions* view. Line 4 is added to the pointcut in order to propagate context information, concerning the class in which the join point is present, to the advice.

Analogously, we declare a view-based pointcut for the *italic output* aspect. We define this pointcut in terms of the *Page Elements* and *Output Generation* views discussed in subsection 5.4:

```
1 classifiedAs(?class,PageElements),
2 send(?joinpoint,?message,?args),
3 withinClass(?joinpoint,?class),
4 classifiedAs(?outputclass,OutputGeneration),
5 methodWithNameInClass(?method,?message,?outputclass)
```

Lines 1 and 2 select all message sends that occur in Wiki page elements. Lines 4 and 5 further restrict these sends to those invoking a method that generates output for the Wiki elements.

Note that both our traditional source-code based pointcut definitions and our view-based pointcut definitions provided a fine-grained description of the actual join points in the program execution that we wish to capture. Our model-based pointcut definitions, however, do not refer to the syntactical or structural organisation of the program on which they act.

5.6 Applying the aspects to the evolved application

To assess the fragility of our view-based pointcuts, we reapply them to the evolved version 1.304 of SmallWiki. However, since view-based pointcuts are defined in terms of intensional views, before reapplying them, we first need to

verify the impact of the evolution on the intensional view model and to try and resolve possible evolution conflicts at that level.

When checking extensional consistency of the *Wiki Actions* view, on which our *logging* pointcut is based, our IntensiVE tool suite warned us that the view had become inconsistent. The feedback provided by the tool informed us that the new actions `save` and `authenticate` did satisfy the second alternative, i.e. they belonged to the `action` method protocol, but they did not adhere to the first alternative, namely their name did not start with `execute`. Also, the tool reported that the `executeSearch` and `executePermission` were not captured by the second alternative, though they did satisfy the first alternative. Note that these inconsistencies match exactly the cases that caused the join point mismatches on our traditional implementation of the *logging* pointcut.

We resolved these mismatches between the model and the source code by performing the following two actions. First, we explicitly declared the `save` and `authenticate` methods as deviating cases to be included in the first alternative of the view. Second, we modified the classification of the `executeSearch` and `executePermission` so that they were correctly classified in the `action` protocol.

It is important to realise that, to detect and resolve these inconsistencies, we did not have to reason about the view-based pointcut itself, but only about the view(s) in terms of which it was defined. Furthermore, after having synchronised the conceptual model with the program again, the pointcut correctly captured the intended join points and we could safely apply the aspect to the code.

The *italic output* aspect depends on two intensional views: *Wiki Page Elements* and *Output Generation*. As for the *logging* aspect, before applying the aspect to the evolved code, we first verified validity of these views with respect to the source code. While the views themselves remained consistent during the considered evolution, our tool suite warned us that the relation *Wiki Page Elements are Accepted By Output Generation* was invalidated. More specifically, it informed us that the relation failed because of the classes `LinkInternal` and `LinkMailTo`. These are exactly the same classes that caused join point mismatches on our traditional implementation of the *italic output* pointcut.

Before applying the *italic output* aspect to the evolved code, we first resolved the conflict. The problem was that the conflicting classes did not have a corresponding visit method in the `OutputVisitor` hierarchy and thus were not directly visited by an output visitor. By adapting the *Output Generation* view so that these classes are explicitly defined as deviating cases to the view, we reconciled the intensional relation with the program. When applying the *italic output* aspect to the code now, it worked as desired.

6 Discussion

Our experiment showed that, when view-based pointcuts are used to implement the *logging of actions* and *italic output* aspects on SmallWiki, the formalism and tool suite of intensional views allowed us to discover exactly those evolution conflicts that caused join point mismatches in a more traditional implementation

of the aspects. To do so, our tool did not reason about the pointcut definitions themselves, but only about the conceptual model in terms of which they were defined. Indeed, because the evolution we applied concerned changes to structure of the base program, only the structural dependencies contained in the conceptual model were affected. After resolving all detected inconsistencies, by synchronising the conceptual model with the evolved program, we could straightforwardly apply the aspects to that program, without having to alter the original view-based pointcut definition.

The core of our contribution lies in the observation that the fragile pointcut problem can be transferred to a problem space where the fundamental cause of the problem (i.e., the structural dependency of pointcuts on code) is isolated and easier to resolve. Rather than addressing the problem at the level of program code, we transfer it to the level of a conceptual model, where extra conceptual information is available that allows us to detect certain join point mismatches. The only requirement is the existence of a conceptual model which allows to express design-level constraints that can be verified against the code, and an aspect language that features model-based pointcuts which can refer to concepts in the conceptual model. Although view-based pointcuts provide a powerful instantiation of model-based pointcuts, one can easily imagine using other models and aspect languages, as we will describe in section 7.

We do not claim that our technique detects and resolves all occurrences of the fragile pointcut problem. Everything depends on the constraints imposed by the conceptual model. Since, in our particular example, we started from a case study which had already been well-documented with intensional views and relations before [10], we were able to detect and avoid all occurrences of the fragile pointcut problem as compared to a traditional AOP approach. In general, the more constraints defined by the conceptual model, the lesser the chance that certain inconsistencies go unnoticed. Further research is required on methodological guidelines to design the conceptual model such that it provides sufficient coverage to detect violations of the design rules.

Adoption of our model-based pointcut approach requires developers to describe a conceptual model of their program and its mapping to the program code. This should not be seen as a burden, because it provides an explicit and verifiable design documentation of the implementation. Such documentation is not only valuable for evolution of aspect-oriented programs but for the evolution of software in general. Providing a means of explicitly codifying and verifying the coding conventions and design rules employed by developers allows them to better respect these conventions and rules. The short term cost of having to design the conceptual model thus pays off on longer-term because it allows keeping the design consistent with the implementation and, consequently, allows detecting potential conflicts when the program evolves.

7 Related and Future Work

In subsection 2.3 we already mentioned some other solutions that have been proposed in the context of the fragile pointcut problem. We now take a closer look at these solutions and describe their differences to our proposed solution. We also describe other closely related work.

Expressive pointcut languages To render pointcut definitions less fragile to base program evolution, more expressive pointcut languages are currently under investigation. The CARMA language, for example, offers a complete logic programming language for the definition of pointcuts. The language features of unification and recursion offer expressiveness to render pointcut definitions more robust [9]. The Alpha aspect language also uses a logic programming language for the specification of pointcuts and enhances the expressiveness by providing diverse automatically-derived models of the program and associated predicates that can, for example, reason over the entire state and execution history [15]. EAOP [14] and JAsCo [20] offer event-based or stateful pointcuts that allow to express the activation of an aspect based on a sequence of events during the program's execution.

Although such expressive pointcut languages permit to render pointcut definitions much less brittle, they do not make the problem disappear altogether. A pointcut definition still needs to refer to specific base program structure or behaviour to specify its join points. This dependency on the base program remains an important source of fragility. To deal with the fragility based on structural dependencies, the user-defined conceptual model, presented in this paper, would even complement the behavioural models used in Alpha and provide additional expressiveness for pointcuts that need to refer to structural properties. Furthermore, one could even argue that, in the occurrence of complex behavioural property-based pointcuts, the rules that the base program needs to respect become very complex to understand. Hence, although more expressive pointcut languages reduce the fragility of pointcut definitions, they may render the actual detection of broken pointcuts more difficult.

Annotations An alternative solution that has been proposed is to define pointcuts in terms of explicit annotations in the code [16, 17]. Similar to intensional views, annotations classify source-code entities and thereby make explicit additional semantics that would otherwise be expressed through implicit programming conventions. This solution, however, addresses the fundamental cause of the problem only partially. While the pointcut definitions are now defined in terms of semantic properties that would otherwise have remained implicit, the problem is displaced to the annotations themselves. Instead of requiring base developers to adhere to implicit programming rules, we now require them to annotate the base program explicitly. As a consequence, pointcuts are as brittle as the annotations to which they refer. When the base code has not been correctly annotated, or when annotations are not correctly updated when the base code evolves, the 'fragile pointcut problem' resurfaces. Havinga et al. [16] try to solve

this problem by inserting the annotations in the code automatically by means of a pointcut that introduces them. However, this again translates the problem to the correctness of that pointcut expression, how well it captures the intention of the aspect developer, and how robust it is towards future changes. Nevertheless, we can easily imagine implementing model-based pointcuts using AspectJ's annotation-property pointcuts, extended with a conceptual model that imposes additional relations and constraints on the annotations that are used.

Pointcut-delta Analysis *Pointcut delta analysis* [7] tackles the fragile pointcut problem by analysing the difference in captured join points, for each pointcut definition, before and after an evolution. The analysis considers statically determinable pointcut deltas and provides a static approximation of the join points which are newly captured or which are no longer captured. A developer can inspect these deltas and verify potential join point mismatches. He is aided in the process because the analysis also states which changes led to the delta.

While this approach can help to assess a number of interesting join point mismatches, accidental misses which result from the addition or modification of source-code entities that *should* be captured by a pointcut, but which are not, are impossible to detect using pointcut-delta analysis. For instance, if we look back at the *logging of actions* pointcut from Section 5, we see that the addition of the `save` method, which is accidentally missed by the pointcut, would not be detected by analyzing the sets of captured join points.

Nevertheless, the ideas proposed in pointcut-delta analysis can be used to create an interesting extension to the model of intensional views. Instead of comparing the sets of the different join points which are captured by the aspects before and after an evolution, we could do a delta analysis on the sets of source-code entities which belong to an intensional view. This way, a developer would be informed of elements which, for instance, change classification or which no longer belong to any classification. Using this feedback, the developer can then (re)classify the source-code entities if needed.

Open Modules, Design Rules and XPI Yet another alternative approach is to explicitly include the pointcut descriptions in the design and implementation of the software and to require developers to adhere to this design. Sullivan et al. [6] propose such an approach by interfacing base code and aspect code through *design rules*. These rules are documented in interface specifications that base code designers are constrained to 'implement', and that aspect designers are licensed to depend upon. Once the interfaces are defined (and respected), aspect and base code become symmetrically oblivious to each others' design decisions. The bare design rules approach does not provide an explicit means to verify if developers adhere to these rules, as opposed to the intensional views model presented in this paper. More recently, the interfaces that are defined by the design rules can be implemented as *Explicit Pointcut Interfaces* (XPI's) using AspectJ [21]. Using XPIs, pointcuts are declared globally and some constraints can be verified on these pointcuts using other pointcuts. Our approach is different in the fact that we keep the pointcut description in the aspect, leaving

more flexibility to the aspect developer. While XPIs fix all pointcut interfaces beforehand, our conceptual model only fixes a classification of the structural source code entities. Another approach is presented by Aldrich in his work on Open Modules [22]. In this approach, modules must advertise which join points can be captured by the aspects that are external to the module. The difference in applicability and expressiveness of these and our approaches remains to be investigated.

Demeter Interfaces. Independently of our work, Skotiniotis et al. [23] address a variant of the fragile pointcut problem, but in the context of adaptive programming, in a way that is very similar to our solution. Adaptive programming is a programming paradigm, akin to aspect-oriented programming, that allows for the separate definition of data structures and traversals over those data structures, with computations attached to the traversal. Current adaptive programming systems provide no mechanisms to warn or guard against modifications that will affect the meaning of a program. To allow for software that is more resilient towards such changes, they introduce *Demeter Interfaces*, which are very similar in spirit to our model-based pointcuts. Demeter Interfaces *decouple* the definition of an Interface Class Graph from the concrete data structure being traversed, thus leaving more flexibility for changes to this data structure. Demeter Interfaces also define structural *constraints* that both the Interface Class Graph and the underlying data structure must satisfy. Finally, the DAJ tool provides support to statically *verify* the mapping of the concrete data structure to the Interface Class Graph as well as the constraints imposed by the Demeter Interfaces.

Conceptual Models Model-based pointcuts alleviate the fragile pointcut problem by specifying pointcuts in terms of a conceptual model. Although in our experiment we opted for the formalism of intensional views to define a conceptual model, we repeat that our approach is independent of the actual formalism chosen. Any formalism which allows the definition of a high-level model of the different concepts in a program, and provides means to keep this model consistent with the program code, can be used as a basis upon which to define model-based pointcuts. A number of such formalisms are the Concern Manipulation Environment (CME) [24], Cosmos [25], Reflexion Models [26], Conceptual Modules [27], ... With minimal effort, that is, given an extended pointcut language that allows to express pointcuts in terms of the concepts in those formalisms, these formalisms could be adopted to provide different flavours of model-based pointcuts, in the like of our 'view-based pointcuts'.

8 Conclusion

The fragile pointcut problem is a serious inhibitor to evolution of aspect-oriented programs. At the core of this problem is the too tight coupling of pointcut definitions with the base program's structure. To solve the problem we propose

the novel technique of *model-based pointcuts*, which translates the problem to a more conceptual level where it is easier to solve. This is done, on the one hand, by decoupling the pointcut definitions from the actual structure of the base program, and defining them in terms of a conceptual model of the software instead. On the other hand, the conceptual model classifies program entities and imposes high-level conceptual constraints over those entities, which renders the conceptual model more robust towards evolutions of the base program. Potential evolution conflicts can now be detected at that level, and solving these conflicts requires changing either the conceptual model or its mapping to the program code, but leaves the model-based pointcut definitions themselves intact.

As a particularly powerful instantiation of model-based pointcuts, we implemented a formalism of *view-based pointcuts*, which extends the CARMA aspect language and combines it with the conceptual model of intensional views. We illustrated the formalism by defining two simple aspects on SmallWiki, an evolving Smalltalk application. When defining the aspects in terms of view-based pointcuts, we managed to discover automatically some instances of the fragile pointcut problem, that went unnoticed when using a more traditional aspect implementation.

9 Acknowledgements

The authors appreciate the feedback received on previous drafts of this paper. In particular, they wish to thank Yann-Gaël Guéhéneuc, Pascal Costanza, Kevin Sullivan, Sebastián González, Tom Tourwé, Mira Mezini and all anonymous reviewers.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtoir, J., Irwin, J.: Aspect-oriented programming. In: European Conference on Object-Oriented Programming (ECOOP). LNCS, Springer Verlag (1997) 220–242
2. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15**(12) (1972) 1053–1058
3. Sommerville, I.: Software Engineering, 6th edition. Pearson Education Ltd (2001)
4. Xu, J., Rajan, H., Sullivan, K.: Understanding aspects via implicit invocation. In: Automated Software Engineering (ASE), IEEE Computer Society Press (2004)
5. Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness (2000) Workshop on Advanced Separation of Concerns (OOPSLA).
6. Sullivan, K., Griswold, W., Song, Y., Chai, Y., Shonle, M., Tewari, N., Rajan, H.: On the criteria to be used in decomposing systems into aspects. In: Symposium on the Foundations of Software Engineering joint with the European Software Engineering Conference (ESEC/FSE 2005), ACM Press (2005)
7. Stoerzer, M., Graf, J.: Using pointcut delta analysis to support evolution of aspect-oriented software. In: International Conference on Software Maintenance (ICSM), IEEE Computer Society Press (2005) 653–656
8. Koppen, C., Stoerzer, M.: Pcdiff: Attacking the fragile pointcut problem. In: First European Interactive Workshop on Aspects in Software (EIWAS). (2004)

9. Gybels, K., Brichau, J.: Arranging language features for more robust pattern-based crosscuts. In: Aspect-Oriented Software Development (AOSD). (2003)
10. Mens, K., Kellens, A., Pluquet, F., Wuyts, R.: Co-evolving code and design with intensional views - a case study. Computer Languages, Systems and Structures **32**(2-3) (2006) 140–156
11. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: International Conference on Software Engineering (ICSE), ACM Press (2005)
12. Mikhajlov, L., Sekerinski, E.: A study of the fragile base class problem. In: European Conference on Object-Oriented Programming (ECOOP). LNCS (1998)
13. Steyaert, P., Lucas, C., Mens, K., D'Hondt, T.: Reuse contracts: Managing the evolution of reusable assets. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'96), ACM Press (1996) 268–285
14. Douence, R., Fritz, T., Loriant, N., Menaud, J.M., Ségura, M., Südholt, M.: An expressive aspect language for system applications with arachne. In: Aspect-Oriented Software Development (AOSD). (2005)
15. Ostermann, K., Mezini, M. Bockisch, C.: Expressive pointcuts for increased modularity. In: European Conference on Object-Oriented Programming (ECOOP). (2005)
16. Havinga, W., Nagy, I., Bergmans, L.: Introduction and derivation of annotations in AOP: Applying expressive pointcut languages to introductions. In: First European Interactive Workshop on Aspects in Software. (2005)
17. Kiczales, G., Mezini, M.: Separation of concerns with procedures, annotations, advice and pointcuts. In: European Conference on Object-Oriented Programming (ECOOP). LNCS, Springer Verlag (2005)
18. Mens, K., Michiels, I., Wuyts, R.: Supporting software development through declaratively codified programming patterns. Special issue of Elsevier Journal on Expert Systems with Applications (2001)
19. Renggli, L.: Collaborative web : Under the cover. Master's thesis, University of Berne (2005)
20. Vanderperren, W., Suvee, D., Cibran, M.A., De Fraine, B.: Stateful aspects in JAsCo. In: Software Composition (SC). LNCS (2005)
21. Griswold, W., Sullivan, K., Song, Y., Shonle, M., Teware, N., Cai, Y., Rajan.H.: Modular software design with crosscutting interfaces. IEEE Software, Special Issue on Aspect-Oriented Programming (2006)
22. Aldrich, J.: Open modules: Modular reasoning about advice. In: Proceedings of the European Conference on Object-Oriented Programming. Volume 3586 of LNCS., Springer (2005) 144–168
23. Skotiniotis, T., Palm, J., Lieberherr, K.: Demeter interfaces: Adaptive programming without surprises. In: European Conference on Object-Oriented Programming (ECOOP). LNCS (2006)
24. Harrison, W., Ossher, H., Jr., S.M.S., Tarr, P.: Concern modeling in the concern manipulation environment. IBM Research Report RC23344, IBM Thomas J. Watson Research Center, Yorktown Heights, NY (2004)
25. Sutton, S., Rouvellou, I.: Modeling of software concerns in cosmos. In: Aspect-Oriented Software Development (AOSD), ACM (2002) 127–133
26. Murphy, G., Notkin, D., Sullivan, K.: Software reflexion models: Bridging the gap between source and high-level models. In: Symposium on the Foundations of Software Engineering (SIGSOFT), ACM Press (1995) 18–28
27. Baniassad, A.L.A., Murphy, G.C.: Conceptual module querying for software reengineering. In: International Conference on Software Engineering (ICSE), IEEE Computer Society (1998) 64–73