

# A Model-driven Pointcut Language for More Robust Pointcuts

Andy Kellens\*  
andy.kellens@vub.ac.be

Kris Gybels  
kris.gybels@vub.ac.be

Johan Brichau  
johan.brichau@liff.fr

Kim Mens  
kim.mens@uclouvain.be

## 1 Introduction

Improved modularity and separation of concerns do not only intend to aid initial development, but are conceived such that developers can better manage software complexity, evolution and reuse [9]. Paradoxically, the essential techniques that AOSD proposes to improve software modularity seem to restrict the evolvability of that software. More specifically, the pointcuts that state when and where aspects need to be invoked during the execution of the base program, are fragile to evolutions of that base program. This is because 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 seriously hamper the evolvability of the software [12]: it implies that all pointcuts of each aspect need to be checked and possibly revised whenever the base program evolves. This problem has been coined *the fragile pointcut problem* [11, 5] and causes evolution problems that are specific to aspect-oriented languages.

We tackle 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 pointcuts* are less likely to change upon evolution, because they are no longer defined in terms of how the program happens to be structured at a certain point in time, but rather in terms of a model of the pro-

gram that is more robust to evolution. Of course, the fragile pointcut problem is now 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[2] combined with the formalism of *intensional views* [6].

## 2 The Fragile Pointcut Problem

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 [11, 5]. The semantics of a pointcut change if the set of join points that is captured by that pointcut changes. We therefore define the fragile pointcut problem as:

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.

---

\*Ph.D. scholarship funded by the “Institute for the Promotion of Innovation through Science and Technology in Flanders” (IWT Vlaanderen).

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**.

## 2.1 Problem Analysis

In general, a pointcut definition makes an assumption about the structure of the base program. 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 [12]). These rules originate from the fact that pointcuts try to define intended conceptual properties about the base program, based on structural properties of the source code. For example, the following ‘accessors’ pointcut tries to define the conceptual property of an ‘accessor method’ by relying on the coding convention that the name of an accessor method starts with `set` or `get`:

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

Consequently, it is required that base program developers adhere to the coding conventions when implementing accessor methods, so that the pointcut definition can be expressed in terms of those rules. However, because these rules imposed by such a pointcut 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 get broken very often, especially upon evolution.

To the best of our knowledge, none of the proposed solutions that exist today (pointcut delta analysis [11], expressive pointcut languages [2, 1, 8], source-code annotations [3, 4], design rules [12]) address *both* the too tight coupling of pointcuts to the structure of the source code, and the brittleness of the imposed design rules with respect to the source code when it evolves. 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.

## 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

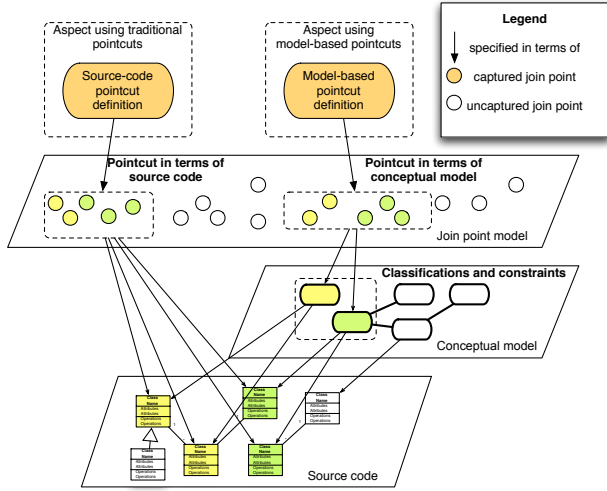


Figure 1: Traditional pointcuts versus model-based pointcuts

structure, model-based pointcuts are significantly less fragile to evolution of the base program because the classifications in the conceptual model are specifically conceived to be more robust to evolution of the base program. For example, assuming that the conceptual model contains a classification of all accessor methods in the base code, the model-based pointcut that captures all call join points to these accessor methods could be defined as:

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

where `classifiedAs(?methSig,AccessorMethods)` matches all methods that are classified as accessor methods in the conceptual model of the base 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 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 [4]. 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 diverted to the level of the conceptual model. Hence, to solve the problem, we still need a mechanism for automatically verifying the correctness of the classifications defined by the conceptual model.

To detect incorrectly classified source entities, the conceptual model goes beyond mere classification (or annotation) and defines extra design constraints that need to be respected by those 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

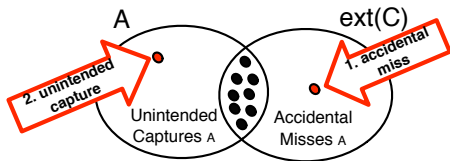


Figure 2: Detecting potential unintended captures and accidental misses

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.

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 are effectively less fragile than source-code based pointcuts because of the following fundamental properties:

- 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 classifies base program entities based on their implementation structure, the specification of the model is enhanced with additional constraints that can be verified to guarantee robustness of the classifications to evolution of the program’s source code.

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 in CARMA

To experiment with model-based pointcuts, we created an extension to the CARMA aspect language that instantiates the conceptual model using the formalism of *intensional views* [6]. We also applied our extension to the implementation of two aspects in the SmallWiki application[10], a fully object-oriented and extensible Wiki framework.

### 4.1 Intensional Views

Similarly to how aspects define the set of join points on which they need to apply their advice, intensional views describe concepts of interest to a programmer by creating views which are groups of program entities (classes, methods, ...) that share some structural property. These sets of program entities are specified intensionally, using the logic metaprogramming language *Soul* [7]. 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]),
methodNameInClass(?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.

Upon evolution of the program, a view 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.

**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.

The IntensiVE tool suite [6] can be used to verify the validity of the constraints, imposed by alternative intensions and 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 CARMA

The CARMA aspect language [2] 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. CARMA also features an open-ended pointcut language. We defined an additional predicate `classifiedAs(?entity,?view)` that allows to define join points in terms of the intensional views defined over a program. For example, the following view-based pointcut captures all calls to methods contained in the `Wiki Actions` view as:

```
pointcut wikiActionCalls():
  classifiedAs(?method,Wiki Actions),
  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 previously introduced robustness of the intensional views model, we can alleviate part of the fragile pointcut problem.

## 5 Discussion

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. 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.

## 6 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 technique of *model-based pointcuts*, which translates the problem to a 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 be detected at that level, and solved by changing either the conceptual model or its mapping to the program code; the model-based pointcut definitions themselves are left intact.

## References

- [1] R. Douence, T. Fritz, N. Lorient, J.-M. Menaud, M. Ségura, and M. Südholt. An expressive aspect language for system applications with arachne. In *4th International Conference on Aspect-Oriented Software Development (AOSD)*, 2005.
- [2] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference of Aspect-Oriented Software Development*, 2003.
- [3] W. Havinga, I. Nagy, and L. Bergmans. Introduction and derivation of annotations in aop: Applying expressive pointcut languages to introductions. In *European Interactive Workshop on Aspects in Software (EIWAS)*, 2005.
- [4] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *European Conference on Object-Oriented Programming, ECOOP 2005*, 2005.
- [5] C. Koppen and M. Stoerzer. Pcdiff: Attacking the fragile pointcut problem. In *First European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.
- [6] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views - a case study. *Computer Languages, Systems and Structures*, 2006.
- [7] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *Special issue of Elsevier Journal on Expert Systems with Applications*, 2001.
- [8] K. Ostermann and C. Mezini, M. Bockisch. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.
- [9] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, December 1972.
- [10] L. Renggli. Collaborative web : Under the cover. Master's thesis, University of Berne, 2005.
- [11] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 653–656, 2005.

- [12] K. Sullivan, W. G. Griswold, Y. Song, Y. Chai, M. Shonle, N. Tewari, and H. Rajan. On the criteria to be used in decomposing systems into aspects. In *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering joint with the European Software Engineering Conference (ESEC/FSE 2005)*. ACM Press, 2005.