Published in the Proceedings of the ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'06), Nantes, France, July 3rd, 2006.

Implementing Bounded Aspect Quantification in AspectJ

Christian Kästner
Department of Computer Science
University of Magdeburg, Germany
kaestner@iti.cs.uni-magdeburg.de

Sven Apel
Department of Computer Science
University of Magdeburg, Germany
apel@iti.cs.uni-magdeburg.de

Gunter Saake
Department of Computer Science
University of Magdeburg, Germany
saake@iti.cs.uni-magdeburg.de

Abstract

The integration of aspects into the methodology of *stepwise software development and evolution* is still an open issue. This paper focuses on the global quantification mechanism of nowadays aspect-oriented languages that contradicts basic principles of this methodology. One potential solution to this problem is to *bound* the potentially *global* effects of aspects to a set of *local* development steps. We discuss several alternatives to implement such bounded aspect quantification in AspectJ. Afterwards, we describe a concrete approach that relies on meta-data and pointcut restructuring in order to control the quantification of aspects. Finally, we discuss open issues and further work.

1. Introduction

Aspect-oriented programming (AOP) aims at localizing, separating, and encapsulating crosscutting concerns [9]. Aspects, the main abstraction mechanism of AOP, modularize those concerns that otherwise would be tangled with and scattered over other concern implementations. While some studies illustrate the success of AOP in several domains, e.g. middleware [6, 18], database systems [16], and operating systems [11], several issues remain controversial or simply not addressed.

This paper aims at the connection of AOP and the methodology of *stepwise software development (SWD)* [17, 15]. The idea behind SWD is to evolve a program from a minimal base by successively applying refinements that encapsulate different design decisions, called *development steps*. This evolutionary process results in a conceptually layered design; each layer implements one refinement and is associated with one development step.

While SWD is fundamental to software development and evolution, it has been shown that the current understanding of AOP does not fit the practice of SWD. Traditionally, aspects affect all elements of a program. This global quantification violates the principle of SWD that refinements must not affect subsequently applied refinements (*local refinement*). This is especially crucial for continuously evolving software. Furthermore, it has been noticed that the current precedence mechanisms of aspects, in particular *AspectJ*[1], are not flexible enough to express different orderings of aspects in layered designs [12, 13].

In prior work we addressed some of these issues: We proposed an architectural model to integrate aspects into layered designs [4]; we presented a concept for understanding aspects as refinements that can be subject to refinement as well [2, 3]. Furthermore, we proposed a mechanism for limiting the effects of aspects to previous development steps [2].

However, the integration of aspects into SWD entails some deeper conceptual and technical issues that remain open. This paper addresses issues regarding the quantification and composition of aspects. Specifically, we present an approach to implement a mechanism for bounding the quantification of aspects so that they fit the practice of SWD. While *bounded quantification* has been discussed theoretically [12, 13], we address several issues that arise from the practical implementation, i.e. in AspectJ. We discuss several alternatives to realize such mechanism and present our experiences and first results. Our work is based on *ARJ*[2], an extended compiler for AspectJ on top of the *AspectBench Compiler*[3] framework.

2. Bounded Aspect Quantification

Traditionally, aspects are quantified globally. That means they may potentially affect all program elements. Unfortunately, this attitude ignores the principle of SWD that refinements are permitted to affect only those refinements that were applied in previous development steps [17, 15]. Several studies have shown that this circumstance is directly responsible for inadvertent aspect interactions and an unpredictable behavior in evolving software [13, 14, 8, 7]

In order to address this issue, Lopez-Herrejon et al. proposed an approach to aspect composition [13]. They model aspects as functions that operate on programs. Applying several aspects to a program is modeled as function composition. In this way the scope of aspects is restricted to a particular step in a program's evolution. Such *bounded quantification* of aspects follows principles of SWD. It has been argued that current AOP languages do not respect this principle because it is not possible to distinguish between different development steps [13, 5].

Suppose the following example: In a first development step we introduce an abstraction for two-dimensional points containing two fields and two setter methods (Fig. 1).

```
1 class Point {
2   int x;
3   void setX(int x){this.x=x;}
4   int y;
5   void setY(int y){this.y=y;}
6 }
```

Figure 1. First step: Introduction of a *Point* class.

In a second step we add an extension for three dimensions and an aspect that counts the updates of *Point* objects (Fig. 2). For that, we introduce a counter variable to *Point* (Line 6) and we intercept and advise executions of setter methods (Lines 7-11). In the present configuration, the *Counter* aspect advises executions of *setX*, *setY*, and *setZ*. Now suppose we apply a further refinement in a subsequent step that introduces a color feature (Fig. 3).

By adding this step, we also affect the counter feature. Although the *Counter* aspect was applied by a previous development step,

[1] http://www.eclipse.org/aspectj/

[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/arj/

[3] http://abc.comlab.ox.ac.uk/

```
1  class Point3d extends Point {
2    int z=0;
3    void setZ(int z){this.z=z;}
4  }
5  aspect Counter {
6    int Point.cntr=0;
7    pointcut setCoordinates(Point p) :
8      execution(* Point*.set*(..)) && target(p);
9    after(Point p) : setCoordinates(p) {
10     p.cntr++;
11   }
12 }
```

Figure 2. Second step: extending the *Point* class and adding a *Counter* aspect.

```
1 class Point3dColor extends Point3d {
2   int Point.color;
3   void setColor(int c){this.color=c;}
4 }
```

Figure 3. Third step: Introduction of a color feature.

it affects the color feature applied subsequently. It advises the *setColor* method and increments the counter of the enclosing *Point* object. But this may not be intended when applying the *Counter* aspect in development step two.

Generally, patterns in pointcuts enable to match a whole bunch of join points and to refine these using one coherent advice. While this is a powerful encapsulation mechanism there are also certain pitfalls, e.g. when code evolves pointcuts may not match anymore [1]. What is interesting for our discussion is that when adding functionality subsequently, such patterns may inadvertently match new join points, as the above example illustrates. Whether this is desired or not in a particular case, it is undesirable for programmers to give up control over these interactions.

One may argue to do not use such fuzzy patterns. But we counter that these mechanisms commonly are considered as an (even though controversial) improvement over other refinement mechanisms [10]. We believe programmers should be encouraged to take advantage of these capabilities, but with certain guarantees, e.g. to affect only things that are currently part of the program. However, we are aware that some concerns are potentially global, e.g. tracing, constraint enforcement, etc. But it has been shown that in principle bounded quantification is able to handle also these global concerns [13].

Our preliminary work on integrating aspects in SWD and layered designs allows for the first time to implement *and experiment with* bounded aspect quantification. This paper presents our ongoing work in this direction. Even if bounded quantification of aspects may be still controversial, our approach may help to prove corresponding arguments and reveal empirical evidence.

3. Preliminary Work

This section reviews our previous results on integrating AOP and SWD that form the basis for this paper.

The idea of SWD is that software is developed and evolved in multiple, sequential steps. Each step refines the program that was developed in previous steps. Aspects are one mechanism to implement such refinements. As mentioned, current AOP languages do not directly support the incremental methodology of SWD. Consequently, we proposed an approach that achieves this: This key idea is that aspects are associated with development steps. Each development step may be associated with several aspects [4, 2, 3].

ARJ is a compiler on top of AspectJ that maintains meta-data about the association of aspects and development steps [3]. One beneficial use of these data is to exploit them for modifying the quantification mechanism: aspects are only allowed to affect aspects of previous development steps. With ARJ, each development step is represented by a distinct directory. A directory may contain several classes and aspects. A configuration file with an ordered list of directory names is used to specify the development steps to be included into the compilation process. Figure 4 shows (a) the directory structure and (b) the configuration file of our example. By mapping steps to directories, the ARJ compiler associates each code fragment with its development step and stores that as meta-data.

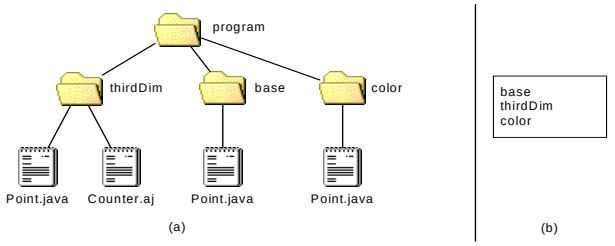

Figure 4. Program organization. (a) File system, and (b) configuration file.

Having this, the idea of bounding aspect quantification can seamlessly be integrated into ARJ: Since the compiler knows for each aspect to which development step it belongs, it can determine to which program elements the aspects are permitted to bind. It uses the meta-data to influence the weaving process.

4. Implementation Alternatives

This section discusses three alternatives of implementing bounded quantification in ARJ: *incremental weaving*, *pointcut restructuring*, and *compiler annotations*.

Incremental weaving. The first approach is to compile the program incrementally. The compilation process starts by compiling the first step and by weaving aspects that belong to this step only. Afterwards, the second step is applied and compiled. Then, aspects associated with that step are woven into the current program consisting now of step one *and* two. Thereby, aspects are automatically limited to the first two steps. The refinements of the remaining steps are applied incrementally in the same manner. Figure 5 illustrates this approach for our example. The *Counter* aspect is woven to the program after the second step. Therefore it affects only the methods *setX*, *setY* and *setZ* that were introduced in the first two steps. Although the pointcut would also match *setColor*, it does not affect the code associated with step three at all.

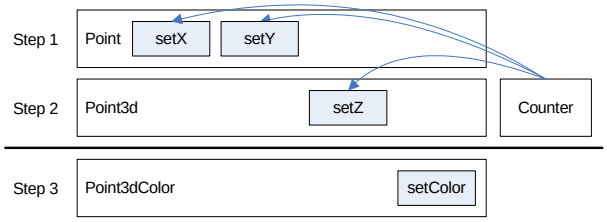

Figure 5. Incremental weaving approach.

Pointcut restructuring. The second approach enforces bounded quantification by restructuring pointcut expressions. The original aspects are modified, so that they do not match join points associated with subsequent steps. By restructuring pointcuts, aspects can be woven into the program in one final step, using any standard AspectJ compiler. As shown in Figure 6, the *Counter* aspect is woven at the end, but still affects only the methods associated with the first two steps, and not *setColor* that fits the pattern, too. This is achieved by excluding those join points from pointcut expressions that are associated with subsequent development steps.

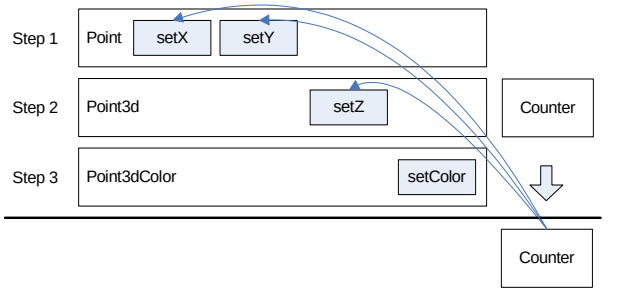

Figure 6. Pointcut restructuring approach.

Compiler annotations. A third approach is to directly extend the AspectJ compiler to bound the quantification of aspects using internal annotations. The compiler's frontend annotates all classes and aspects with information about the associated development steps. During the weaving process the compiler's backend uses these annotations to match permitted join points only. For this approach the compiler's frontend and the pointcut matcher must be adapted. This approach does not produce source code as a separate step but directly weaves the aspects into the program. The aspect quantification is bounded directly during the weaving process.

Discussion. All of the considered approaches have advantages and disadvantages. The incremental weaving approach is very complex. It changes the whole compilation process, so that the program is compiled in multiple steps. It also requires major changes to the AspectJ compiler to disable the existing support for advice precedence and to cope with semi-woven classes. It technically enforces bounded quantification very directly and consequently and without the need of source code analysis. All in all this approach is solid but requires major compiler changes and makes the compilation process very complex.

The pointcut restructuring approach is not trivial either. To restructure an aspect's pointcuts, it is necessary to analyze all potential target join points (its shadows) in each development step to determine their scope. This presumably requires to modify parts of the compiler's frontend and the pointcut matcher. The benefit of this approach is that a source-to-source conversion is possible. The resulting source code can be compiled with any AspectJ compiler. This helps the programmer to get insight into the restructured code. In contrast to the incremental weaving approach, it is not necessary to change the compilation process or to work with semi-woven class files. Additionally this approach is more flexible since it is possible to implement transformations that allow defined exceptions from the bounded quantification (cf. Sec. 6). Such exceptions cannot be implemented with the incremental weaving approach because the strict bounding is enforced technically by the weaving process.

The third approach annotates the code and extends the compiler with an altered pointcut matcher. The approach is similar to pointcut restructuring, but bounded quantification is enforced in the compiler's backend, instead of the frontend. Therefore, a source-to-source transformation is not possible. Furthermore, also a static program analysis for annotation is requited. The approach offers a similar flexibility as pointcut restructuring. However, it lacks transparency for the programmer.

We choose to implement the pointcut restructuring approach in ARJ because its flexibility and transparency are vital for the programmer and for further language extensions. This allows us to experiment with exceptions from the strict bounded quantification approach and to quickly change the restructuring algorithm. In ongoing work, we will consider also the alternative approaches. For now, we limit our considerations to pointcut restructuring.

5. Bounded Aspect Quantification in ARJ via Pointcut Restructuring

Pointcut restructuring can be implemented completely in the frontend of the ARJ compiler. It uses the available meta-data that map code fragments to development steps.

5.1 Mechanisms for Pointcut Restructuring

We use two principle mechanisms to restrict pointcuts. The first replaces pointcut patterns by method signatures (*wildcard replacement*). This way, it can be ensured that a pointcut cannot accidentally match fitting methods introduced in later development steps. This requires a complete static program analysis for each step. However, every step reuses that information from previous steps.

Figure 7 shows one possible version of a restructured *Counter* aspect. In this transformed version all pattern expressions have been replaced by fully qualified method signatures ("*Point*.set*(..)*" was replaced with *Point.setX*, *Point.setY* and *Point3d.setZ*). Thus, it matches only *setX*, *setY* and *setZ* that were introduced in the first two development steps, and not *setColor* introduced in the third step.

```
1 aspect Counter {
2   pointcut setCoordinates(Point p):
3     (execution(void Point.setX(int))
4     || execution(void Point.setY(int))
5     || execution(void Point3d.setZ(int)))
6     && target(p);
7   ...
8 }
```

Figure 7. Restructured Counter aspect.

The second mechanism uses *within* pointcuts to restrict the pointcut matcher to classes associated with certain steps (*within constraints*). The *within* pointcut matches classes with a certain type pattern. It can be used to restrict an existing pointcut with pattern expressions to one or more specific classes.

Figure 8 shows the restructured *Counter* aspect when using this mechanism. In this example, two *within* pointcuts have been added to restrict the pattern expression of the *execution* pointcut to *Point* and *Point3d*. It is not necessary to modify the original pointcut pattern (Line 3). The *within* pointcut can also be used to restrict pointcuts that match the client side. i.e. *get*, *set*, and *call*. In this case, pointcuts were restricted to all possible, permitted client classes.

We believe that by combining wildcard replacement and within constraints it is possible to cover a wide range of transformations necessary to enforce bounded aspect quantification.

After the process of pointcut restructuring, the compiler can proceed in two ways (Fig. 9). Either (a) weaves the transformed aspects directly to the target classes or (b) it writes the modified aspect sources out. The sources can be used to compile the program with an external AspectJ compiler or just for debugging purposes.

```
1 aspect Counter {
2   pointcut setCoordinates(Point p):
3     execution(* Point*.set*(..)) && target(p)
4     && (within(Point) || within(Point3d));
5   ...
6 }
```

Figure 8. Restructured aspect using the *within* pointcut.

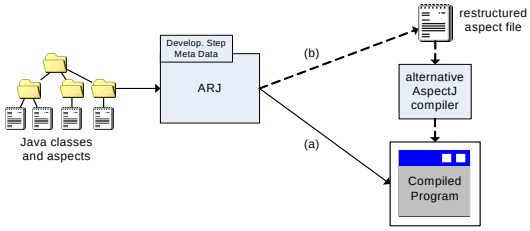

Figure 9. ARJ compilation process.

5.2 Pointcut Semantics in AspectJ

In the following, we examine some selected pointcuts in the light of pointcut restructuring for implementing bounded quantification.

During our attempts to implement bounded quantification in ARJ we realized that the semantics of pointcuts in AspectJ are not really defined precisely. Moreover, even between different compiler versions (*ajc version 1.2 vs. 1.5*) and different vendors (*ajc vs. abc*), we found minor variations in the semantics. For our analysis we refer to the semantics that can be experimentally determined from the ajc compiler version 1.5.

We limit our discussion to *execution*, *call*, *set*, and *get* because they are the most commonly used ones and they reveal some open issues. To illustrate the problems, we modify our running example as shown in Figure 10. We add a *Draw3D* class to the second step that instantiates and uses the *Point3d* class and that is extended in the third step by *Draw3dColor*. Additionally, the method *setY* is extended by *Point3dColor* in the third step.

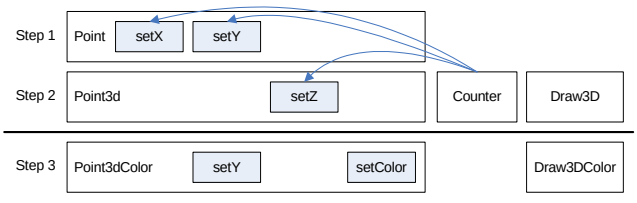

Figure 10. Extended Example.

Execution pointcuts match method executions depending on the type of the target class. Therefore, it is necessary to specify the exact target type in which the method is defined. It is not possible to define an *execution* pointcut matching a method inherited from a super class. Instead the method must be literally defined in the target class. In our initial example in Figures 1, 2, and 3 the pointcut "*execution(* Point3d.setX(..))*" would not match because the *setX* method is not defined or extended in the *Point3d* class. In contrast, *call*, *get* and *set* pointcuts match the client side that calls the method or accesses the field. They match methods and fields either defined in the target class or inherited from super classes. Therefore the pointcuts "*call(* Point.setX(..))*" and "*call(* Point3d.setX(..))*" both match the calls from *Draw3d* to this method.

5.3 Semantics of Bounded Quantification

Execution pointcuts. *Execution* pointcuts are least problematic with regard to bounded quantification. An *execution* pointcut is already bounded to the target class and thereby to a single development step. Hence, *execution* pointcuts bear no potential for unexpected effects on subsequent development steps, unless the target class is specified with a pattern expression. In such cases, as shown in our example in Figure 7, the pattern expression is reduced to match target classes from early development steps only. For the developer the reduced scope of target class pattern expressions is the only change to the semantics of *execution* pointcuts. This change is intuitive and follows the semantics of bounded quantification.

Call pointcuts. In contrast to *execution* pointcuts, *call* pointcuts match the client side of a method invocation, i.e. the caller. The target object is not directly affected. In our example, the advice code would be woven into the *Draw3D* class. This causes two problems that might result in unexpected effects:

First, the *call* pointcut has to match only calls to methods that already were introduced in the development step that the aspect belongs to. In our example the *Counter* aspect – with a *call* instead of an *execution* pointcut – would only match calls to the *setX* method that were invoked from the initial version of the *Draw3D* class. Calls from the subclass *Draw3dColor* (third step) are not advised because this extension has been added in a subsequent step. This might surprise developers at first, but is explained with the basic principle of bounded quantification. To match all calls to a method independently of the step where the call originates from an *execution* pointcut may be used.

The second effect occurs when the target method itself is extended. The advice is woven into the caller, independently whether the target is extended or overridden in a subsequent step. In our example, the *Counter* aspect – with a *call* instead of an *execution* pointcut – matches the *setY* calls from the *Draw3D* class, even though the *setY* method is extended later in the third step. Depending on the extension this may again lead to unforeseen behavior in some rare cases, namely when the pointcut matches a call to a method changed in later steps. However, this is not a specific problem of AspectJ in the context of SWD, but a general issue about virtual methods calls. Nevertheless, due to the pointcut restructuring approach the ARJ compiler is aware of those situations and may issue warnings or even limit or change the strict requirements for bounded quantification.

We propose not to modify the *call* pointcut semantics but to explicitly document these possible effects. Furthermore, we suggest to evaluate in detail the cases where *call* pointcuts can be used in SWD and to adapt the pointcut restructuring process accordingly.

Get and set pointcuts. The *get* and *set* pointcuts are woven into the program at the caller side, similar to *call* pointcuts. Therefore the same problem occurs as described for *call* pointcuts: The client that accesses a field value must already exist in the development step where the aspect is added. In contrast to *call* pointcuts, where it is possible to switch to *execution* pointcuts, there is no equivalent alternative that matches the access to a field at the receiver.

Therefore, we argue that strict bounded quantification limits the usability of *get* and *set* pointcuts. It could be useful to introduce a pointcut type to AspectJ that matches field access join points on the target side, similar to the *execution* pointcut for methods, e.g., by treating field accesses and assignments as functions. An alternative approach is the introduction of a controlled possibility to specify unbounded aspects, as suggested in Section 6.

Other Pointcuts. Now, we give an overview over further selected pointcuts. The *within* pointcut matches one target class. Similar to *execution* pointcuts, *within* pointcuts are modified only when

pattern expressions are used; other bounding mechanisms are not necessary. The *initialization* pointcut is woven into the target class similar to *execution* pointcuts and therefore can be handled equally. The *this*, *target* and *args* pointcuts do not need any change. They bear no possible side effects for SWD. These pointcuts only match the type of the caller, the target, or the arguments. Pattern expressions are not allowed.

The *handler* pointcut is woven in all exception handlers matching a given type pattern. Pattern expressions must be reduced as described for *execution* pointcuts to not match new exception types added in subsequent development steps. As with *get* and *set* pointcuts, *handler* pointcuts are woven only into exception handlers that already existed in the development step where the aspect is defined. Still a typical use case for *handler* pointcuts is to globally modify the handling of a certain exception type. We therefore again suggest the introduction of global pointcuts (see Sec. 6).

The pointcut *cflow* and related pointcuts do not match any basic join points themselves, but are used to further specify other pointcuts. It therefore does not introduce any new problems. Still, when using *cflow* pointcuts the developer must keep in mind the semantic changes of the other pointcut types.

6. Discussion and Conclusion

The integration of AOP into the methodology of stepwise software development and evolution promises various benefits, but also requires bounding the quantification of aspects. Aspects are not allowed to influence join points associated with subsequent development steps.

In this paper, we presented a mechanism to implement bounded aspect quantification in ARJ by restructuring pointcut expressions to match only join points that are permitted to advise. Pointcut restructuring promised higher flexibility and transparency than alternative approaches, i.e. incremental weaving or annotations.

Bounded aspect quantification is supposed to avoid inadvertent effects by reducing the number of possible interactions between development steps [12, 13, 5]. However, the developer might want to add global, unbounded aspects to the program. Examples are global constraint enforcement, tracing, profiling, etc. As illustrated in Section 5.3, *set* and *get* pointcuts would benefit from a less strict bounded quantification. We suggest to evaluate the necessity for global aspects and possible language mechanisms that integrate bounded and unbounded aspects, e.g. via a *global* keyword for aspects, pointcuts, or advice.

For the ARJ project the integration of single aspects with bounded quantification into incrementally developed classes is only a first step. ARJ supports *mixin-based inheritance* for classes and aspects themselves, as well as *aspect refinement*, *pointcut refinement* and *advice refinement* [3]. Bounding the quantification when working with refined classes and refined aspects induces new issues: A composite aspect, evolved over several development steps, is associated with these multiple steps. This makes it necessary to determine which parts of the aspect are bound to which step.

Furthermore, mixin-based inheritance introduces a new problem to determine the pointcut's actual target, especially for *execution* pointcuts, because a target class may consist of multiple refinements associated to multiple development steps. Finally, the semantics of pointcut refinement and advice refinement themselves require deeper evaluation. They enable advanced opportunities to restrict or extend aspects in later development steps. Discussions must emphasize usability and comprehensibility from the developers point of view, to make the effects of refined aspects predictable and avoid inadvertent effects. For these extensions the high flexibility and transparency of the pointcut restructuring approach is vital. Our long term goal is to fully integrate aspects into the methodology of SWD and layered designs.

Acknowledgments. This work was done while Sven Apel was visiting the group of Don Batory at the University of Texas at Austin. It is is sponsored in parts by the German Research Foundation (DFG), project number SA 465/31-1, as well as by the German Academic Exchange Service (DAAD), PKZ D/05/44809.

References

[1] T. Tourwe abd J. Brichau and K. Gybels. On the Existence of the AOSD-Evolution Paradox. In *AOSD Workshop on Software Engineering Properties of Languages for Aspect Technologies*, 2003.

[2] S. Apel, T. Leich, and G. Saake. Aspect Refinement and Bounding Quantification in Incremental Designs. In *Proceedings of Asia-Pacific Software Engineering Conference*, 2005.

[3] S. Apel, T. Leich, and G. Saake. Mixin-Based Aspect Inheritance. Technical Report 10, Department of Computer Science, University of Magdeburg, Germany, 2005.

[4] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of International Conference on Software Engineering*, 2006.

[5] S. Apel and J. Liu. On the Notion of Functional Aspects in Aspect-Oriented Refactoring. In *Proceedings of ECOOP Workshop on Aspects, Dependencies, and Interactions*, 2006.

[6] A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2004.

[7] R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *Proceedings of Generative Programming and Component Engineering*, 2002.

[8] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2004.

[9] G. Kiczales et al. Aspect-Oriented Programming. In *Proceedings of European Conference on Object-Oriented Programming*, 1997.

[10] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[11] D. Lohmann et al. A Quantitative Analysis of Aspects in the OS Kernel. In *Proceedings of ACM SIGOPS EuroSys Conference*, 2006.

[12] R. Lopez-Herrejon and D. Batory. Improving Incremental Development in AspectJ by Bounding Quantification. In *AOSD Workshop on Software Engineering Properties and Languages for Aspect Technologies*, 2005.

[13] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 2006.

[14] N. McEachen and R. T. Alexander. Distributing Classes with Woven Concerns: An Exploration of Potential Fault Scenarios. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2005.

[15] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2), 1979.

[16] A. Tesanovic et al. Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software. *Journal of Embedded Computing*, October 2004.

[17] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4), 1971.

[18] C. Zhang and H.-A. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages and Applications*, 2004.