RAM-SE'06 - ECOOP'06 Workshop on Reflection, AOP, and Meta-Data for Software Evolution

(Proceedings)

Nantes, 4th of July 2006

Edited by

Walter Cazzola - Università degli Studi di Milano, Italy
Shigeru Chiba - Tokyo Institute of Technology, Japan
Yvonne Coady - University of Victoria, Canada
Gunter Saake - Otto-von-Guericke-Universität Magdeburg, Germany

Preprint no. 13/2006 of the Faculty of Computer Science, Otto-von-Guericke-Universität Magdeburg.

Foreword

Software evolution and adaptation is a research area, as also the name states, in continuous evolution, that offers stimulating challenges for both academic and industrial researchers. The evolution of software systems, to face unexpected situations or just for improving their features, relies on software engineering techniques and methodologies. Nowadays a similar approach is not applicable in all situations e.g., for evolving nonstopping systems or systems whose code is not available.

Reflection and aspect-oriented programming are young disciplines that are steadily attracting attention within the community of object-oriented researchers and practitioners. The properties of transparency, separation of concerns, and extensibility supported by reflection and aspect-oriented programming have largely been accepted as useful for software development and design. Reflective features have been included in successful software development technologies such as the Java language and the .NET framework. Reflection has proved to be useful in some of the most challenging areas of software engineering, including Component-Based Software Development (CBSD), as demonstrated by extensive use of the reflective concept of introspection in the Enterprise JavaBeans component technology.

Features of reflection such as transparency, separation of concerns, and extensibility seem to be perfect tools to aid the dynamic evolution of running systems. They provide the basic mechanisms for adapting (i.e., evolving) a system without directly altering the existing system. Aspect-oriented programming can simplify code instrumentation providing a few mechanisms, such as the join point model, that permit of evincing some points (*join points*) in the code or in the computation that can be modified by weaving new functionality (aspects) on them in a second time. Meta-data represent the glue between the system to be adapted and how this has to be adapted; the techniques that rely on meta-data can be used to inspect the system and to dig out the necessary data for designing the heuristic that the reflective and aspect-oriented mechanisms use for managing the evolution.

It is our belief that current trends in ongoing research in reflection, aspect-oriented programming and software evolution clearly indicate that an inter-disciplinary approach would be of utmost relevance for both. Therefore, we felt the necessity of investigating the benefits that the use of these techniques on the evolution of object-oriented software systems could bring. In particular we were and we continue to be interested in determining how these techniques can be integrated together with more traditional approaches to evolve a system and in discovering the benefits we get from their use.

Software evolution may benefit from a cross-fertilization with reflection and aspect-oriented programming in several ways. Reflective features such as transparency, separation of concerns, and extensibility are likely to be of increasing relevance in the modern software evolution scenario, where the trend is towards systems that exhibit sophisticated functional and non-functional requirements; that are built from independently developed and evolved COTS (commercial off-the-shelf) components; that support plug-and-play, end-user directed reconfigurability; that make extensive use of networking and internetworking; that can be automatically upgraded through the Internet; that are open; and so on. Several of these issues bring forth the need for a system to manage itself to some extent, to inspect components' interfaces dynamically, to augment its application-specific functionality with additional properties, and so on. From a pragmatic point of view, several reflective and aspect-oriented techniques and technologies lend themselves to be employed in addressing these issues. On a more conceptual level, several key reflective and aspect-oriented principles could play an interesting role as general software design and evolution principles. Even more fundamentally, reflection and aspect-oriented programming may provide a cleaner conceptual framework than that underlying the rather 'ad-hoc' solutions embedded in most commercial platforms and technologies, including CBSD technologies, system management technologies, and so on. The transparent nature of reflection makes it well suited to address problems such as evolution of legacy systems, customizable software, product families, and more. The scope of application of reflective and aspect-oriented concepts in software evolution conceptually spans activities related to all the phases of software life-cycle, from analysis and architectural design to development, reuse, maintenance, and, therefore also evolution.

The overall goal of this workshop – as well as of its previous editions – was that of supporting circulation of ideas between these disciplines. Several interactions were expected to take place between reflection, aspect-oriented programming and meta-data for the software evolution, some of which we cannot even foresee. Both the application of reflective or aspect-oriented techniques and concepts to software evolution are likely to support improvement and deeper understanding of these areas. This workshop has represented a good meeting-point for people working in the software evolution area, and an occasion to present reflective, aspect-oriented, and meta-data based solutions to evolutionary problems, and new ideas straddling these areas, to provide a discussion forum, and to allow new collaboration projects to be established. The workshop is a full day meeting. One part of the workshop will be devoted to presentation of papers, and another to panels and to the exchange of ideas among participants.

In this third edition of the workshop, we had an interesting keynote by Awais Rashid on relation among aspects and evolution. This keynote was an interesting experiment that has raised several issues and lively discussion among the workshop attendees. To the interested reader an extended abstract can be found in the first part of these proceedings.

This volume gathers together all the position papers accepted for presentation at the third edition of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'06), held in Nantes on the 4th of July, during the ECOOP'06 conference. We have received many interesting submission and due to time restrictions and to quality insurance we had to choice few of them, the papers that, in our opinion, are more or less evidently interrelated to feed up a more lively discussion during the workshop. Now, few months after the workshop, we can state that we achieved our goal, presentations were interesting and the subsequent panels grew up lively and rich of ideas and proposals. We are sure that in the next months we will see many papers by the workshop attendees and fruit of such a lively discussions.

The success of the workshop is mainly due to the people that have attended it and to their effort to participate to the discussions. The following is the list of the attendees in alphabetical order.

Altman, Rubén	Dubochet, Gilles	Pini, Sonia
Bernard, Emmanuel	Eaddy, Mark	Raibulet, Claudia
Beurton-aimar Marie	Ebraert, Peter	Rashid, Awais
Cámara Moreno, Javier	Horie, Michihiro	Saake, Gunter
Cazzola, Walter	Kästner, Christian	Shakil Khan, Safoora
Chiba, Shigeru	Masuhara, Hidehiko	Stein, Krogdahl
Cyment, Alan	Meister, Lior	Südholt, Mario
David, Pierre-Charles	Nguyen, Ha	Tsadock, Carmit
D'Hondt, Theo	Pérez Toledano, Miguel Ángel	Zambrano, Arturo

A special thank is for the three chairmen (Theo D'Hondt, Hidehiko Masuhara, and Mario Südholt) that governed the panels at the end of each session.

We have also to thank the Department of Informatics and Communication of the University of Milan, the Department of Mathematical and Computing Sciences of the Tokyo institute of Technology and the Institute für Technische und Betriebliche Informationssysteme, Otto-von-Guericke-Universität Magdeburg for their various supports.

October 2006 W. Cazzola, S. Chiba, Y. Coady, and G. Saake
RAM-SE'06 Organizers

Contents

Keynote on Aspects and Evolution

Aspects and Evolution: The Case for Versioned Types and Meta-Aspect Protocols. . 3
Awais Rashid (Computing Department, Lancaster University, UK).

Aspect-Oriented Modeling for Software Evolution

Improving AOP Systems' Evolvability by Decoupling Advices from Base Code. . . . 9
Alan Cyment, *Nicolas Kicillof*, *Rubén Altman*, and *Fernando Asteasuain*
(University of Buenos Aires, Argentina).

Making Aspect Oriented System Evolution Safer. 23
Miguel Á. Pérez Toledano, *Amparo Navasa Martinez*,
Juan M. Murillo Rodriguez (University of Extremadura, Spain)
Carlos Canal (University of Málaga, Spain).

Design-Based Pointcuts Robustness Against Software Evolution. 35
Walter Cazzola (DICo, University of Milan, Italy),
Sonia Pini and *Massimo Ancona* (DISI, University of Genova, Italy).

Tools and Middleware for Software Evolution

Evolution of an Adaptive Middleware Exploiting Architectural Reflection. 49
Francesca Arcelli and *Claudia Raibulet*
(Università degli Studi di Milano-Bicocca, Italy).

An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution. . . 59
Javier Cámara Moreno, *Carlos Canal*, *Javier Cubo* (University of Málaga, Spain)
Juan M. Murillo Rodriguez (University of Extremadura, Spain).

An Aspect-Aware Outline Viewer. 71
Michihiro Horie and *Shigeru Chiba* (Tokyo Institute of Technology, Japan).

Technological Limits for Software Evolution

Solving Aspectual Semantic Conflicts in Resource Aware Systems. 79
Arturo Zambrano, *Tomás Vera* and *Silvia Gordillo*
(University of La Plata, Argentina),

Statement Annotations for Fine-Grained Advising. 89
Marc Eaddy and *Alfred Aho* (Columbia University, USA).

Dynamic Refactorings: Improving the Program Structure at Run-time. 101
Peter Ebraert and *Theo D'Hont* (Vrije Universiteit Brussel, Belgium).

Implementing Bounded Aspect Quantification in AspectJ. 111
Christian Kästner, *Sven Apel*, *Gunter Saake*
(Otto von Guericke University Magdeburg, Germany).

Aspects and Evolution: The Case for Versioned Types and Meta-Aspect Protocols

Keynote speaker: Awais Rashid, Lancaster University, UK

Chairman: Shigeru Chiba, Tokyo Institute of Technology, Japan

Aspects and Evolution: The Case for Versioned Types and Meta-Aspect Protocols

Awais Rashid

Computing Department, Infolab21, Lancaster University, Lancaster LA1 4WA, UK
awais@comp.lancs.ac.uk

One of the often cited advantages of aspect-oriented programming (AOP) [4] is improved evolvability. It is often suggested that *quantification* and *obliviousness*, as proposed by Filman and Friedman [3], are the key properties that facilitate the high degree of evolvability in AOP systems. However, quantification and obliviousness are only desirable properties of AOP systems [9]. Filman and Friedman talked about "better AOP systems" being "more oblivious" as well as of "incomplete obliviousness". Several application studies of AOP, e.g., Kienzle and Guearroui [5], Rashid and Chitchyan [6] and Fabry [2], have revealed that in many cases obliviousness is undesirable and can, in fact, be harmful. Fabry [2], in fact, distinguishes between syntactic and semantic obliviousness and points out that syntactic obliviousness can be achieved, however, semantic obliviousness can neither be achieved nor desirable. Similarly, though quantified statements do help match multiple join points in a pointcut expression, quantification is not the essence of AOP. It is simply one possible aspect composition mechanism albeit a popular one. Filman and Friedman talked about "interfaces" between advice and base action. Similarly, Colyer et al. [1] discuss *heterogeneous* advice where difference pointcuts (with associated advice) capture a single join point but together they form a coherent concern.

So if obliviousness and quantification are not fundamental properties of AOP, then why are aspects good for evolution? The much more fundamental properties of AOP: *abstraction*, *modularity* and *composability*, as highlighted by Rashid and Moreira [9], are what make aspects good for evolution. *Abstraction* allows us to abstract away from the details of how that aspect might be scattered and tangled with the functionality of other modules in the system and, in turn, abstract away from unwanted details of the change. *Modularity* allows us to reason about changes to a crosscutting concern in isolation and realise those changes with minimal ripple effect. Finally, *composability* allows us to reason about the global or emergent properties of an aspect-oriented system, facilitates propagation of necessary changes and guards against propagation of unwanted changes.

No doubt, this abstraction, modularity and composability support for crosscutting concerns helps to localise changes thus supporting evolution. However, evolution often requires keeping track of changes in order to make them reversible. Furthermore, often such changes (and their reversal) needs to be done online, e.g., in case of business and mission critical systems that can't be taken offline. This requires

first class support for versioned types as well as fully-fledged meta-aspect protocols. Such a versioned type system and a meta-aspect protocol have been experimented with in the VEJAL aspect language and its associated dynamic evolution framework [7, 8], which provides a high degree of dynamic adaptability of object database evolution strategies. We need to build on the VEJAL experience to investigate how versioned type semantics may be incorporated into mainstream AOP languages. Furthermore, the notion of a meta-aspect protocol requires reconsideration of aspect composition models – we need to move away from syntactic dependencies on the base elements and instead focus on join point models derived from the semantics of the application domain (as is the case for the VEJAL join point model for the object persistence domain). Furthermore, the design of such a meta-aspect protocol needs to consider relevant flexibility vs openness trade-offs, e.g., as studied by Welch and Stroud in the context of meta-object protocols and Security [10].

In summary, aspect-oriented software development research can incorporate complementary techniques from software evolution and the work on reflection and meta-object protocols to develop techniques that are more capable of supporting aspect evolution, both statically and at runtime, resulting in aspects that are more resilient to changes.

Acknowledgments. The work on VEJAL discussed in this extended abstract was conducted by Nick Leidenfrost for his MPhil thesis (Lancaster University) conducted as part of the UK Engineering and Physical Science (EPSRC) Research Grant: AspOEv: An Aspect-Oriented Evolution Framework for Object-Oriented Databases (GR/R08612), 2000-2004. The author also wishes to thank Gordon Blair (Lancaster University, UK) and Adrian Colyer (Interface21) for discussions on the notion of a meta-aspect protocol as well as Ana Moreira (New University of Lisbon, Portugal) and Ruzanna Chitchyan (Lancaster University, UK) on fundamental software engineering properties of aspect-oriented systems. The author is supported by European Commission Grant: AOSD-Europe: European Network of Excellence on Aspect-Oriented Software Development (IST-2-004349), 2004-2008.

References

[1] A. Colyer, A. Rashid, and G. S. Blair, "On the Separation of Concerns in Program Families", Computing Dept., Lancaster University Technical Report COMP-001-2004 (http://www.comp.lancs.ac.uk/computing/aose/papers/COMP-001-2004.pdf) 2004.

[2] J. Fabry, "Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code": PhD Thesis, Vrije Universiteit Brussel, Belgium, 2005.

[3] R. Filman and D. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", OOPSLA WS on Advanced Separation of Concerns, 2000.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", European Conference on Object-Oriented Programming (ECOOP), 1997, Springer-Verlag, Lecture Notes in Computer Science, 1241, pp. 220-242.

[5] J. Kienzle and R. Guerraoui, "AOP: Does It Make Sense? The Case of Concurrency and Failures", European Conference on Object-Oriented Programming (ECOOP), 2002, Springer-Verlag, Lecture Notes in Computer Science, 2374, pp. 37-61.

[6] A. Rashid and R. Chitchyan, "Persistence as an Aspect", 2nd International Conference on Aspect-Oriented Software Development, 2003, ACM, pp. 120-129.

[7] A. Rashid and N. Leidenfrost, "Supporting Flexible Object Database Evolution with Aspects", International Conference on Generative Programming and Component Engineering (GPCE), 2004, Springer-Verlag, Lecture Notes in Computer Science, 3286, pp. 75-94.

[8] A. Rashid and N. Leidenfrost, "VEJAL: An Aspect Language for Versioned Type Evolution in Object Databases", AOSD Workshop on Linking Aspect Technology and Evolution, 2006.

[9] A. Rashid and A. Moreira, "Domain Models are NOT Aspect Free", Proceedings of MoDELS/UML, 2006, Springer, Lecture Notes in Computer Science, 4199, pp. 155-169.

[10] I. S. Welch and R. J. Stroud, "Re-engineering Security as a Crosscutting Concern", *The Computer Journal, Special Issue on Aspect-Oriented Programming and Separation of Crosscutting Concerns (To Appear)*, No., 2003.

Aspect-Oriented Modeling for Software Evolution

Chairman: Theo D'Hondt, Vrije Universiteit Brussel, Belgium

Improving AOP systems' evolvability by decoupling advices from base code[1]

Alan Cyment, Nicolas Kicillof, Rubén Altman and Fernando Asteasuain

Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires
{acyment, nicok, raltman, fasteasuain}@dc.uba.ar

Abstract. The evolvability of AOP systems is severely affected by the tight coupling between aspects and base code. This paper identifies the advice fragility problem, originated in the need for advices to access the application context while being oblivious to base code details. Most proposed solutions to the well-known pointcut fragility problem consist in decoupling base code from aspects by means of an intermediate abstraction layer. We build on top of those proposals, introducing the concept of model-based aspects, and present a new version of our semantic pointcut framework, constituting a practical approach to address the advice fragility problem.

1 Introduction

Since the very inception of the AOSD family of concepts and technologies, the degree to which so-called aspect-oriented systems have successfully coped with the woes of software evolution has been thoroughly scrutinized [1]. One of the most debated ideas is the obliviousness concept [3], first heralded by the community as a core prerequisite for considering a system truly aspect-oriented, but lately bashed by several authors [4, 23] as an obstacle to the successful evolution of AOSD systems.

Most of the existing research on this subject has so far focused on the pointcut fragility problem (coined by [2], and referred to by this and other names in [5, 6, 8, 9, 12]), which basically describes the dangerous coupling between an oblivious base code and a given pointcut descriptor (PCD) that heavily relies on the low-level structure of that code. Nevertheless, not much attention has yet been paid to the problem of maintaining *advices* synchronized with an evolving base code. Changing a method name can easily break advice code which relies on the structure of the class that has evolved (i.e., if no aspect-aware refactoring aides are used). This issue will hereafter be referred to as the *advice fragility problem*. Following AspectJ conventions, we use the term "aspect" to denote the combination of advices and pointcuts. Hence, we will also refer to the *aspect fragility problem* to describe the compound issue posed by the two fragility problems described so far.

[1] This work was partially funded by Microsoft Research's Phoenix—Excellence in Programming RFP Awards and ANCyT PICT 11738

Proposed solutions to the pointcut fragility problem generally put forward mechanisms to decouple base code from aspects through the definition of an intermediate abstraction layer [4, 9, 10, 11]. We extend those proposals by introducing the concept of *model-based aspects*. These are basically pointcuts and advices that, instead of relying on the low-level structure of base code, are decoupled from it by being defined in terms of an intermediate, more abstract, conceptual description of the domain modeled by the application. The question of whether fragility is totally eradicated or just shifted remains open, but we aim at shedding some light on the subject by analyzing different ways to represent and map the abstract layer.
The next section characterizes the advice fragility problem; section 3 presents our proposal for dealing with this problem; section 4 puts this proposal into perspective by showing our implementation of these ideas and a concrete example; the remaining sections conclude the work.

2 Aspect Fragility = (Advice + Pointcut) Fragility

The *obliviousness* principle, originally considered by AOSD pioneers as one of the most rewarding features of this emerging technology, has recently been "considered harmful" by some researchers [4, 23]. Keeping base code completely unaware of the existence of aspects would undoubtedly alleviate the task of developers of the former. But recent research has shown that the natural evolution of base code will very easily wreak havoc with the aspect side of the equation. Using existing AOP tools, if an aspect programmer were to follow the obliviousness principle, she would be forced to tightly couple her pointcuts to the base code, in order to completely avoid adapting the base code to the aspects.
But pointcuts are not the only component of aspects. We claim that mechanisms that implement the obliviousness principle tend to also make *advices* excessively coupled to base code. Consequently, when base code evolves, they tend to become obsolete too. Advice is essentially code written in a programming language. Its goal is to model, at least in part, the behavior of a given domain, which happens to be a crosscutting concern of the complete application under development. However, advice is not autonomous: it will only be executed at a joinpoint (i.e. a given point during the lifetime of the base code). Due to the *quantification* principle [3], advice code must be built so that it can successfully interact with heterogeneous base code. If the latter evolves (i.e. changes its structure), there is an evident risk of the original advice code to cease fulfilling the implicit contract intended by its programmer. We thus state that the fragility of advices lies in the way they *access context*.
There are different ways in which advices access context in AspectJ-like tools. In all cases, the evolution of base code can bring about errors in the complete system: whenever base code changes, programmers are to review all existing advice code in order to avoid these errors. The resulting coupling of advice and base code results in what we have termed advice fragility.
One form of context access is what we term *pointcut signature*, it is sometimes called "typed advice" or "pointcut parameter" in the literature. The advice receives objects

of certain types as arguments, which are provided by the weaver. A source of coupling in this process is that advice code often expects these objects to be in a given state (e.g. already initialized) or to be related to one another in a certain way (e.g. the first parameter is always smaller than the second one). Moreover, evolution of base code may result in the need to change the type of parameters of both a pointcut and its associated advice. Another mechanism used by many AOP tools to access base-code context in advices is *reflection*. If base code changes, it is very easy for reflective advice code to, for example, try to instantiate and invoke a non-existent method.

3 Model-based aspects

Most proposed solutions to the pointcut fragility problem [9, 12, 21, 22] decouple PCDs from base code by making the former depend not on the low-level structure of the latter, but rather on an intermediate, more abstract layer that aims at describing the problem at hand in a more domain-oriented fashion than raw code. Following this idea, we claim that this layer must be conceived in such a way that advice code can benefit from the resulting decoupling as well. This is illustrated in Figure 1.

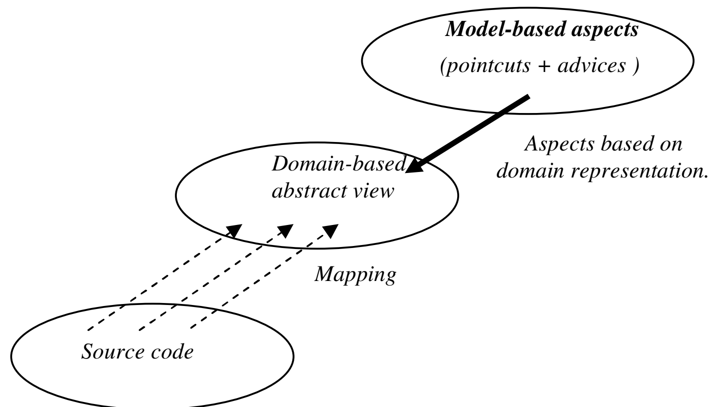

Fig. 1. Model-based Aspects

As a running example, we will analyze an application from an architectural point of view. The system consists of two components, connected via an event handler. Our model must provide a representation for each concept in the architectural view, and also a means to make relationships between these concepts explicit, as they comprise the reification of the architecture's configuration. Also, a mapping must be established between fragments of base-code and the concepts they represent.

3.1 Relating base code and model

Mechanisms to relate the base code with the abstract layer can be broken down into those based on predicates (i.e. over base code) [4, 9] and those using annotations [12, 22]. Predicates are used for establishing contracts that base code ought to comply with, and are by definition intensional (i.e. they are written once, but define a subset of the entire base code). Annotations (i.e. metadata), on the other hand, are each associated to a single code entity, and can therefore be considered extensional. Predicates are thus a better way of achieving the quantification property [3].

If context is to be accessed in a conceptually higher level of abstraction, advice code should be able to collaborate with living objects that represent entities from the conceptual model. Object-oriented developers work by defining classes and instances, and only in some flavors of the paradigm do they use invariants or global contracts, which makes annotations a more intuitive choice. Nevertheless, annotations, due to their extensional nature, are more sensitive to changes in base code.

Clearly neither of the options is completely satisfying on its own. We have chosen to develop our first prototype using annotations due to our background in object-oriented development, but we consider the best choice to be a combination of both approaches, which we plan to implement in future versions of our framework.

3.2 The need for mappings

As mentioned before, advice fragility is caused by context access from advice code. Context access can be thought of as the need for advices to collaborate with (or refer to) objects originated in the base-code world. In contrast to pointcuts, where coupling to base code stems from the PCDs relying on base-code structure and execution flow, advices rely both on the naming and semantics (i.e. contract fulfillment) of base-code methods.

Probably the most widely adopted way of decoupling collaboration among objects is the adapter pattern [24]. The basic function of an adapter is to stand in the middle of a given collaboration, so as to decouple both ends. In the same fashion, a series of patterns, namely façade and mediator, decouple an object from a group of other objects that share a responsibility. Considering the common features of these patterns, we claim that the intermediate layer should allow a *mapping of both behavior and structure between base code and the world of aspects*. Using object-oriented models for the abstract layer clearly offers a clean way of specifying this mapping. [10] and [11] use a similar idea to decouple collaboration between advice and base code.

Back to our model architecture, let us suppose that, at base-code level, events are identified with numerical IDs, but at the architectural description level they are named using mnemonic strings. This non-trivial mapping can easily be achieved by defining an ad-hoc mapping object.

3.3 Comparing explicit and implicit models

For simple applications, conceptual models need not be explicitly defined in a separate document. When this happens, annotations will refer to concepts that are not defined elsewhere, they merely become loose labels. This is a practical solution, provided there are no relevant relationships between concepts, and only when mappings from base-code elements to concepts are straightforward.

As an example of an implicit conceptual model, let us assume a developer wants to enrich base code with a trivial model that classifies methods into those that update data and those that retrieve it. Using this simple model, the developer could later write a simple transaction aspect that enlists only updating methods. Later on, a caching aspect could be added, in order to cache the returned value from all methods that retrieve data. No added value would result from specifying the data updating/retrieval model in some knowledge-representation language, e.g. by defining an abstract "data operation" concept specialized by both kinds of methods. There is no relevant relationship between concepts in the model (i.e. they are orthogonal) and no mapping behavior needs to be attached to them (i.e. they easily map to methods in base code on a one-to-many basis).

On the other hand, a more complex scenario, like the architectural view in our running example, calls for making the conceptual model explicit. In this case, there are relevant relationships between model entities, and a translation from base-code elements to conceptual entities needs to be specified.

As we have seen, these needs arise for most moderately complex systems. Consequently, we have chosen explicit over implicit models for our framework. It must be mentioned though, that implicit models are preferred when possible, due to the lower system and conceptual overhead they cause.

4. PROOF-OF-CONCEPT IMPLEMENTATION

Continuing with the work introduced in [14], the SetPoint tool for .NET has been extended by implementing the ideas presented in the previous section. SetPoint was one of the first AOP tools to explicitly attack the pointcut fragility problem. The context access problem was mentioned in [14] as future work, which naturally led to adding support for conceptual context access in the new version of the tool.

SetPoint was developed using the C# language. It works in two steps. First, assemblies are preprocessed by injecting code to allow method call interception. Then, during runtime, the SetPoint engine (an assembly itself), analyzes each method call (i.e. join point) to determine if it belongs to any of the declared pointcuts (see [13] for a similar approach). This new version of SetPoint is being developed using Microsoft's Phoenix framework [25].

4.1 Intermediate model

SetPoint uses object-orientation as the knowledge representation formalism for the intermediate conceptual model (see [26] for a comparison with a different formalism). Together with code annotations, they constitute the mechanism for decoupling aspects from base code.

The model consists of what we call *concepts*. We represent them with C# interfaces. There are basically two kinds of concepts: *actions* and *entities*. Entities have attributes called *entity properties*; actions, in turn, have *action roles* (which can be played by entities or instances of other .NET types). Entities stand for concepts in the application domain. Actions represent system-wide events in the model. They are the conceptual equivalent of join points, allowing aspect developers to refer to points in the flow of a program in a more abstract way, decoupled from low-level implementation details. The interfaces that model entities and actions merely define their entity properties and action roles, respectively.

All transformations from base code to the model are made by *mappers*: C# objects that implement the above-mentioned interfaces. *Action mappers* implement, in terms of low-level join points,.NET properties that represent action roles. Low-level join points are constructs that reify runtime events, such as method calls or constructor calls: they contain the message sender, receiver, arguments and the selector (following pure object oriented terminology). *Entity mappers*, in turn, wrap base-code objects or groups of objects. They therefore have access to all their public members, in order to implement getters and setters for properties in the corresponding interface.

Lastly, code annotations, implemented with .NET attributes (stored as program metadata), relate base-code elements (such as methods or classes) to the model. Attributes are actually special classes, so whole attribute hierarchies can be defined, with different behavior for each of their members. As we did with mappers, we have chosen to differentiate entity and action annotations.

5. EXAMPLE

We will use a reduced banking application as a simple example. The application domain can informally be specified as follows (bracketed words and phrases correspond to semantic concepts):

There are [accounts].
Each [account] has an [account number].
An [account number] is a [string].
Each [account] has a [balance].
A [balance] is a [rational number].
There are [operations] that [update] an [account]'s [balance].
Each [operation] has a [name].
A [name] is a [string].
A [savings account] is a kind of [account]

The requirement that we intend to implement as an aspect is conceptually defined as follows:

> *After an [operation] that [updates] a [savings account]'s [balance]; if the resulting [balance] is negative then write in the system event log the [account number], the [operation]'s [name] and the [balance].*

As said, the domain is modeled using interfaces. This means that, each entity or action is represented with a .NET interface. As an example, we show here the interface that conceptually represents an operation that modifies an account's balance:

```
interface IBalanceUpdate : IAction {
          public IAccount Account { get; }
          public string Name { get; }
          public float Amount { get; }
}
```

Similarly, the following couple of interfaces specify the savings account concept:

```
interface IAccount : IEntity {
          public string AccountNumber {get;}
          public float Balance {get;}
}

interface ISavingsAccount : IAccount { }
```

Interface IBalanceUpdate has three .NET properties to represent action roles, namely an account, a name and a balance, as specified in the informal domain description. Interface ISavingsAccount has two properties: its number and its balance (both inherited from interface IAccount). The mapping from a base code to the resulting model is implemented via annotations and mappers. A possible base-code scenario is the following:

```
class SA {
          float Balance;
          string AccountNumber;
          void Withdraw (float Amount) {…}
          …
}
```

The simplest way provided by SetPoint to map this class to its conceptual counterpart is to make class SA to directly implement interface ISavingsAccount. This would require adding appropriately named.NET properties to the class. A less intrusive mechanism is to create a mapper and associate it to the class through an entity annotation. The following code shows an example for the latter alternative:

```
[EntityAnnotation("ISavingsAccount", "SAMapper")]
class SA {…}
```

Withdraw is an operation that updates an account's balance. It must therefore be annotated, so as to relate it with the corresponding concept in the abstract model. The annotation's attribute must state that this is an action of type IBalanceUpdate and also select the mapper to be used for abstraction. Entities that correspond to this action's roles must also be annotated. Roles Account and Amount can be directly annotated in the base-code: the instance receiving message Withdraw is to be mapped to role Ac-

count; role Amount can be mapped by annotating the Amount parameter of the aforementioned method:

```
class SA {...
  [ActionAnnotation(
    "IBalanceUpdate", "BalanceUpdateMapper"
  )]
  [ReceiverRoleAnnotation("IBalanceUpdate", "Account")]
  void Withdraw(
    [RoleAnnotation("IBalanceUpdate", "Amount")]
    float Amount
  ) {…}
  …
}
```

Using annotations and conceptual definitions as input, SetPoint synthesizes the C# code that implements mappers, automatically generating mapping behavior when the necessary information is available (when it is not, the developer must manually supply the mapping code). In this case, SetPoint automatically constructs the BalanceUpdateMapper class using the information provided by the action annotation. Meanwhile, role annotations provide the information to generate direct mappings for the Account and Amount properties. On the other hand, no annotation has been given for property Name, which must thus be manually implemented:

```
class BalanceUpdateMapper:JoinPoint, IBalanceUpdate{

  public IAccount Account{
    get{
      //Automatically generated
      return new SAMapper(this.Receiver);
    }
  }

  public string Name{
    get{
      //Manually added
      return this.Message.Name;
    }
  }

  public string Amount{
    get{
      //Automatically generated
      return this.GetParameterValue(1);
    }
  }
}
```

Action mappers are subclasses of the JoinPoint class. This class abstracts low level context-access knowledge such as receiver, sender and message.

So far, we have shown how the domain model is defined and how elements in base code are mapped to conceptual entities and actions. We next explain how aspects interact with these structures by defining a pointcut, an advice, and the corresponding

aspect. It's worth mentioning at this point that we have chosen to use a familiar and known AspectJ-like notation since SetPoint's notation is still under refinement.

```
aspect BalanceLogging{

  pointcut SavingsAccountBalanceUpdate(
    out IBalanceUpdate bu
  ){
      Action(bu) && Role[Account](ISavingsAccount)
  }

  after(IBalanceUpdate bu):
    SavingsAccountBalanceUpdate(bu){
      ValidateSABalance(bu)
  }

 void ValidateSABalance (IBalanceUpdate bu){
    if(bu.Account.Balance < 0){
      Debug.WriteLine(
        String.Format(
          "After operation {0}
          the balance of account {1} is {2}",
          bu.Name,
          bu.Account.AccountNumber,
          bu.Account.Balance
        )
      );
    }
  }
}
```

Pointcut SavingsAccountBalanceUpdate establishes that whenever a message annotated as performing a balance modification action is reached, the corresponding mapper will be instantiated. This aspect is what we have termed model-based aspect, since both pointcuts and advices are defined in terms of a conceptual domain representation. Context access (in this case, account balance update information) is specified at a high level of abstraction, alleviating the advice fragility problem presented before.

Let us assume now that base code evolves, and in a future version the account number is not directly represented by a field in class SA, but obtained as the concatenation of two fields, namely CustomerID and AccountSuffix. All we would need to do is adapt the corresponding property getter in class SAMapper (the adapter between base code and model). Both pointcut and advice would remain oblivious to this change, alleviating the aspect fragility problem.

6. RELATED & FUTURE WORK

6.1 Context access

Reflection is used as the mechanism for context exposure and composition of PCDs in frameworks like Josh [5]. But the required metaprogramming skills may become quite complex. In this respect, [17] makes an interesting point about the relationship between AOP and reflection: AOP engines should be built on top of reflection libraries, so that metaprogramming becomes intuitive. Our approach follows this line, thus we use reflection, but we avoid exposing its inherent complexity to the aspect programmer.

6.2 Intermediate Layer

Other approaches also intend to solve the pointcut fragility problem by increasing the expressive power and abstraction level of pointcuts [6, 8, 9], but the lack of a more abstract, semantic view than the base code itself greatly limits the power of the proposed solutions.

The Model-Based Pointcuts approach [9] also advocates for relying on an intermediate abstract layer in order to decouple aspects from base code. In this model, source-code entities that address the same concern are grouped together in views using logic programming mechanisms such as predicates, instantiation and pattern matching. AOP technology can be used on top of this model, making PCDs less dependent on the low-level structure of base code. However, these views are generated directly from the base-code syntax, rather than live in the semantic world (i.e. domain representation) as in SetPoint. Our approach makes evolution much easier because it embodies a higher level of abstraction.

We are not aware of any work that has yet focused on what we have called the advice fragility problem.

6.3 Other metadata approaches

The authors of Compose*[16], from the Composition Filters school, propose the use of metadata entries to tag base code elements with design information. The main difference with our approach is that this proposal offers no structure for metadata: there are no relationships between tags, so no complex domain model can be expressed.

The notion of collaborations, roles, and composition of different views is also exploited in the collaboration-based design approach Object Teams [11]. A new kind of collaboration module called Team is introduced to capture multi-object collaboration. This new module basically combines properties of packages and classes, containing inner classes where each element implements a role in the collaboration. Teams are composed from the base code by binding each of their roles in collaboration to a base

class through an explicit mapping. In SetPoint, it is also possible for two or more base-code entities to collaborate to form a single concept in the semantic world with a proper mapping.

More similar to SetPoint, but in a different context, Tuna [18] refers to tags that are part of a model that lets the annotator make use of knowledge-representation semantics. Instead of AOP, metadata is used in this case to enrich program semantics, so as to, according to the authors, bridge the gap that exists between MDA and the XP development methodology. In the same spirit, Chris Welty's PhD dissertation [19] presents a source code ontology [20], which should allow maintenance coders to more easily browse an application they had never seen before, leveraging code entities annotated by original developers using this ontology.

6.4 Entities and mappers

There are several works introducing the use of interfaces in order to obtain more reusable and general aspects [4, 10, 15]. In [10], aspect implementation and aspect binding are specified in different modules, and interfaces glue them together to form reusable aspects. These interfaces, called collaborative-interfaces (CI), specify what aspects provide to the context in which they are applied, and also what aspects expect from that context. Although CIs help in decoupling base code from aspect code, they do not constitute an intermediate layer as SetPoint has. That is to say, interfaces in SetPoint define a semantic world, playing a totally different role in the development process.

6.5 Future work

The next challenge for SetPoint is to keep refining the interfaces, mappers and annotations model. We will continue to investigate the expressive power of interfaces as a domain representation. We are also improving automatic mapper generation and making it possible for object models already defined at the base code level to be used as part of the conceptual level without any mediating annotation.

To the best of our knowledge, there is currently no AOP tool based on an intermediate layer that uses both predicates and metadata for relating base code to the abstract model. We think a combined approach could be beneficial and are therefore considering the possibility of adding contract-like predicates to our modeling formalism.

Work on a full-scale example will help us delve deeper into the question of whether fragility is merely shifted or effectively reduced when an intermediate abstraction layer is used between aspects and base code.

8. CONCLUSION

Decoupling aspects from base code is crucial to improve aspect-oriented software evolution. In this respect, several approaches claim the need for aspects to refer to

base code from an intermediate, more abstract point of view. Keeping this in mind, we had developed the first version of SetPoint, where the abstract view was based on a representation of the domain. However, this version was not expressive enough to accommodate context access in the domain-based abstract view, as the model did not allow behavior mapping. As a consequence, it was not suitable for addressing the advice fragility problem. In order to overcome these limitations, we here present a new version of SetPoint, where we propose a practical approach to this problem. A more complex mapping mechanism consisting of annotations and mappers provides a domain-based abstract view with context-access capabilities. In this model, not only do pointcuts rely on this view, but advices too, resulting in what we call model-based aspects.

References

1. P.Tarr, M.D'Hont, L.Bergmans and C.V.Lopes. Requirements on, and Challenge Problems For, Advanced Separation of Concerns. Workshop on Aspects and Dimensions of Concern. ECOOP 2000.
2. C.Koppen and M.Stoerzer. Pcdiff: Attacking the fragile pointcut problem. EIWAS 2004
3. R.Filman and D.Friedman. Aspect-oriented programming is quantification and obliviousness. Advanced Separation of Concerns. OOPSLA 2000.
4. W.G. Griswold, K.Sullivan, Y.Song, M.Shonle, N.Tewari, Y.Cai and H.Rajan et al. Modular Software Design with Crosscutting Interfaces. IEEE Software, vol. 23, no. 1, pp. 51-60, Jan/Feb. 2006.
5. S.Chiba and K.Nakagawa. Josh: An Open AspectJ-like Language. AOSD 2004.
6. M.Eichberg, M.Mezini and K.Ostermann. Pointcuts as Functional Queries. APLAS 2004.
7. K.Gybels and J.Brichau. Arranging Language Features for More Robust Pattern--Based Crosscuts. AOSD 2003.
8. H.Masuhara and K.Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. APLAS 2003.
9. A.Kellens, K.Mens, J.Brichau, and K.Gybels. Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. ECOOP 2006.
10. M.Mezini and K.Ostermann. Conquering Aspects with Caesar. AOSD 2003.
11. S.Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. Proc. of International Conference NetObjectDays. 2002.
12. R.Altman, A.Cyment and N.Kicillof. On the need for SetPoints. EIWAS 2005.
13. R.Douence and M.Sudholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes. 2002.
14. R.Altman and A.Cyment. SetPoint: a semantic approach for the pointcut resolution in AOP. Msc. Thesis, Universidad de Buenos Aires. 2004.
15. G.Kiczales and M.Mezini. Aspect-oriented programming and modular reasoning. ICSE '05.
16. C.Noguera García. Compose * A Runtime for the .Net Platform. Msc. Thesis, University of Twente, 2003.
17. G.T.Sullivan. Aspect-oriented programming using reflection and meta-object protocols. Comm. ACM, 44(10):95–97, 2001.
18. C.Zimmer and A.Rauschmayer. Tuna: Ontology-Based Source Code Navigation and Annotation. Workshop Ontologies as Software Engineering Artifacts in OOSPLA 2004.

19. C.Welty. An Integrated Representation for Software Development and Discovery. Ph.D. Thesis, Rensselaer Polytechnic Institute. 1996.
20. T.Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. International Workshop on Formal Ontology, 1993.
21. W.Cazzola, S.Pini and M.Ancona. Evolving Pointcut Definition to Get Software Evolution. RAM-SE'04-ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution. 2004.
22. I.Nagy, L.Bergmans, W.Havinga and M.Aksit. Utilizing Design Information in Aspect-Oriented Programming. Proc. of International Conference NetObjectDays, NODe2005. 2005.
23. C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral sub typing analogy. Technical Report TR03-01a, Iowa State University. 2003.
24. E.Gamma, R.Helm, R.Johnson and J.Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley. 1994.
25. http://research.microsoft.com/phoenix
26. A.Cyment, N.Kicillof and F.Asteasuain. Enhancing model-based AOP with behavior representation. Second Workshop on Models and Aspects – ECOOP 2006. 2006.

Making Aspect Oriented System Evolution Safer[a]

Miguel A. Pérez Toledano[1], Amparo Navasa Martínez[1],
Juan M. Murillo Rodríguez[1], Carlos Canal[2].

[1] University of Extremadura (Spain), Department of Computer Science,
Quercus Software Engineering Group,
{toledano, amparonm, juanmamu}@unex.es.

[2] University of Málaga (Spain), Department of Computer Science,
GISUM Group,
canal@lcc.uma.es

Abstract. The information systems of enterprises change rapidly and it is necessary for the existing software to evolve without the comprehension, modularity or quality of the built systems being affected. In this context, Aspect Oriented Programming reveals as an adequate way of working because it makes the encapsulation of methods easier and reduces development times. The inclusion of aspects (woven code) inside an existing software code could, nevertheless, cause the resulting system behaviour not to be that expected by the developer. In this paper, we propose to evolve system specifications at the same time as the software itself. In this way, we can start from these specifications in order to obtain state machines and algebraic descriptions of the components of the system. These can be used to perform verification, simulation or testing operations of the systems built. We also propose the use of extended state machines that allow us to describe the evolution of the system in a more detailed way, facilitating more complete Model Checking operations.

Keywords: Aspect Oriented System Evolution, Extended State Machines, Algebraic Specifications, Validation, Verification, Simulating, Model Checking.

1 Introduction

The information systems of enterprises change rapidly and it is necessary for the existing software to evolve without the comprehension, modularity or quality of the built systems being affected. The use of Aspect Oriented Programming (AOP) facilitates this task, allowing us the encapsulation of methods that, otherwise, would be scattered in the code of the different software elements of the system. This encapsulation makes the comprehension of the code easier, reduces the time of system development and, on the whole, allows software systems to evolve rapidly. The evolution of the systems in which woven aspects exist and the difficulty to access to

[a] Research supported by CICYT Project (number TIN-2005-09405-C02-02)

the source code of these applications could, nevertheless, produce a series of problems, described in [1], which could then cause the final system behaviour not to be that expected by the developer.

Currently, there is not tool that allows a complete study of every possible problem when aspects are included inside a system. Existing papers in this area are focused on some of the possible situations. Some works study how the addition of new aspects could affect a system (and also which are the properties affected by the new woven code), by means of code analysis or static Model Checking [2] and are based on the use of state machines or algebraic descriptions of the system components, while other proposals study the increased system (analysing the woven code and comparing its properties with the underlying software system) using tools to check the resulting code as Bandera [3] and Java Pathfinder [4].

Moreover, in order to generate quality software and to document the built system it is necessary to achieve a previous detailed Analysis and Design of the system to build. When systems are object-oriented, the Unified Modelling Language (UML) is usually employed as the modelling tool [5]. Several papers use UML as a tool to model aspects and to add its behaviour inside the system to build [6].In [7], state charts are used to describe aspects behaviour and also to integrate this behaviour into the state charts that describe the joint points associated. Other works, like [8], separate the specification of aspects behaviour from the system business rules. For this, the use of sequence diagrams to describe aspectual scenarios, with Interaction Patterns Specifications (IPS) [9] is proposed. These IPS will subsequently be instantiated inside those system sequence diagrams that describe the associated joint points. The objective is to obtain state charts from system components and use them later to achieve simulation and validation operations with the existing requirements.

In this paper we propose to study the integration of aspects into a software system starting from its UML specifications. This study will be focused, nevertheless, on the interactions described by sequence diagrams, assuming that the modelling will probably affect also other types of diagrams, which are beyond the scope of this paper.

As regards the contribution of this paper, we propose to build more complete state machines than the currently used state charts [7,10]. These extended machines will allow us to represent information about time counters, about fragments (as used in UML 2.0) and also about system variables. All this information will permit to achieve more precise model checking operations of the system. Moreover, we study the effects of adding new aspects inside a system before they have been woven and, furthermore, to study the behaviour of the woven code. In order to facilitate this study, a technique for grouping components is proposed, so that simulation traces can be reduced due to the omission of internal interactions among grouped components, focusing on the interaction with the surrounding environment.

This paper is structured into the following points: some of the problems derived from integrating aspects inside a system are described in Section 2; Section 3 describes our proposal; while an example is presented Section 4; Section 5 contains the conclusions and future works.

2 Problem Description

The construction of software systems using Java makes the use byte code possible, protecting in this way the source code of classes. This originates that the evolution of these systems by means of AOP lacks precise information about the underlying system in which aspects must be applied. Because of that, errors can be caused when languages as AspectJ [11] are used in order to weave aspects inside Java code. These problems [1] can be summarized in:

– Unintended aspects effects**.** When pointcuts of new aspects of the underlying system are applied, they may be applied to undesired joint points of classes, and this could provoke unintended side effects.

– Arbitrary aspect precedence**.** When pointcuts of new aspects of the underlying system are applied, they may be applied to the same joint point as other (unknown) aspects already are. This may cause problems with the sequence of application of aspects

– Unknown aspect assumptions**.** When pointcuts of new aspects of the underlying system are applied, they may not find joint points matching existing requirements.

– Partial weaving**.** When the code of a system is modified, the aspects inside it may not be applied to future modifications.

These problems are caused by the difficulty of knowing the existence of previous woven aspects inside the code and the ignorance of the pointcuts defined in them.

3 Proposal

As discussed in the previous section, the evolution of Java systems by means of AOP is hindered by the frequent ignorance of the source code. In this context, the creation of an adequate specification of the system and its later adequate use to evolve the system are necessary. The following steps are proposed in order to document the system (graphically described in Figure 1):

1. System modelling by means of UML. It is necessary to separate the aspect description from the rest of the system. The idea consists of describing the behaviour of aspects in such a way that, when the system evolves, a complete documentation about its behaviour exists. This paper is focused on the study of interaction diagrams but the specification of aspects must be described inside every affected UML diagram. Interaction Patterns Specifications (IPS) will be used in order to describe aspect interactions. This tool is based on describing patterns that represent the expected interactions of the aspect to be integrated by means of sequence diagrams [8].
2. Instantiating aspect patterns (IPS) inside the sequence diagram of the system. We need to find points in the sequence diagram matching the requirements stated by aspects description where aspects can be introduced. Note that

different operation exist in order to instantiate the patterns, depending on the description of the aspects. It is possible to find further information about design and instantiation of IPS in [12].

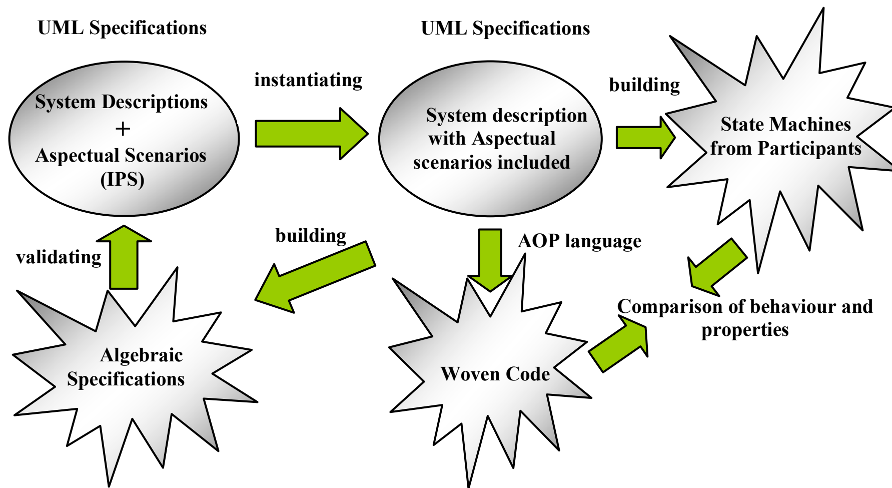

Fig. 1. Building of system specifications steps

3. The information described during system modelling is frequently not precise enough, as it is necessary to describe the system completely. In order to detect errors and gaps, the obtained specifications must be validated [13].For that purpose, algebraic descriptions of the specifications obtained are built and model checking is applied to detect deadlocks and inconsistencies. This point will allow us to complete specifications in such a way that, once an error is detected, it is necessary to return to point number one to solve it.

4. Once specifications have been validated, extended state machines can be automatically obtained for each element of the system. There exist several algorithms to achieve this task [14]. In this paper, we will use the algorithm proposed in [15]. It consists of obtaining a state machine for each scenario in which a system is involved and then machines are assembled. State labels, described in sequence diagrams, are usually employed to assemble machines in order to identify the state in which the component is inside the scenario. Nevertheless, the machines obtained [18] provide more precise descriptions than statecharts.

5. The specifications must be renewed to evolve the system. In order to do that, it is necessary to return to point number 1 and update documentation. Once new behaviours have been described, instantiating each aspect inside the system into adequate joint points is again necessary. This will allow us to detect problems in the sequence of execution of aspects and to prove if there exist adequate joint points in which aspects can be applied. Finally, aspects must be woven inside

the code again in order to avoid partial weaving problems. Once the whole process is finished, new machines are obtained. Its behaviour will reflect the new described behaviour. However, it is still possible that problems arise when proving if the behaviour of the obtained woven code agrees with design.

Extended state machines describe the planned behaviour of the system and the woven code describes the final behaviour obtained. Studying possible unintended effects of the aspects inside the system will consist of studying if state machines and code behaviour match. In order to achieve this task, model checking techniques can be used. These operations must study the properties of both systems and must complete the study by means of simulating the same execution traces in both of them. To execute model checking operations inside Java code obtained, Java Pathfinder can be used, whereas for state machines, UPPAAL tool is proposed [16]. UPPAAL is an integrated tool environment for modelling, simulation and verification of systems. It is appropriate for systems that can be modelled as a collection of processes with finite control structure and real-valued clocks, communicating through channels or shared variables. The UPPAL simulator enables examination of possible dynamic executions of a system during modelling stage and, thus, provides an inexpensive means of fault detection prior to the verification by the model-checker. The UPPAAL model-checker covers the exhaustive dynamic behaviour of the system; it can also check invariant and reachability properties by exploring the state-space.

To compare machines and code simulation, two alternatives exist:

- Generating traces from UPPAAL. These traces simulate the execution of machines and can be used as inputs into the generated code.
- Weaving one aspect inside the code that does not modify the behaviour and be limited to monitor the code execution and create traces able to be used in the UPPAAL simulator.

This second option seems to be more elegant because it permits the use of AOP concepts in order to study systems built by means of AOP. Nevertheless, execution traces obtained may be too large due to the size of the built systems. In order to reduce size, it is achievable to group components and to monitor only the interesting events to facilitate the simulation and to focus the study on the attractive points. Components grouping [17] allow us to obtain descriptions suited to sets of components, abstracting from internal interactions among them. These groups will be adapted to developer needs and will allow us to create traces exclusively containing the events of interests.

4 Example

Two scenarios are presented in order to illustrate our proposal (Figure 2). The first one represents the behaviour of one aspect, depicted by means of an IPS. The second one represents one scenario of the system that achieves the necessary requirements to apply the mentioned aspect. In order to prove if a certain aspect is applicable to a

given scenario, we must check the associated restrictions, described by means of state labels. In the example, to instantiate the aspect, it is necessary to establish some binds between the role "|rd:" and the element "c2:class4", and between the method "|notif()" and the method "operat1()". Once the aspect is composed, the specification of the system is available. Notice that the type of instantiation of the aspect will be executed depending on its description and there exists several operations to achieve it [12]. Once final specifications have been obtained, algebraic descriptions are built with CCS, and Model Checking analysis is performed for detecting errors and deadlocks. The specifications will be modified until they are correct.

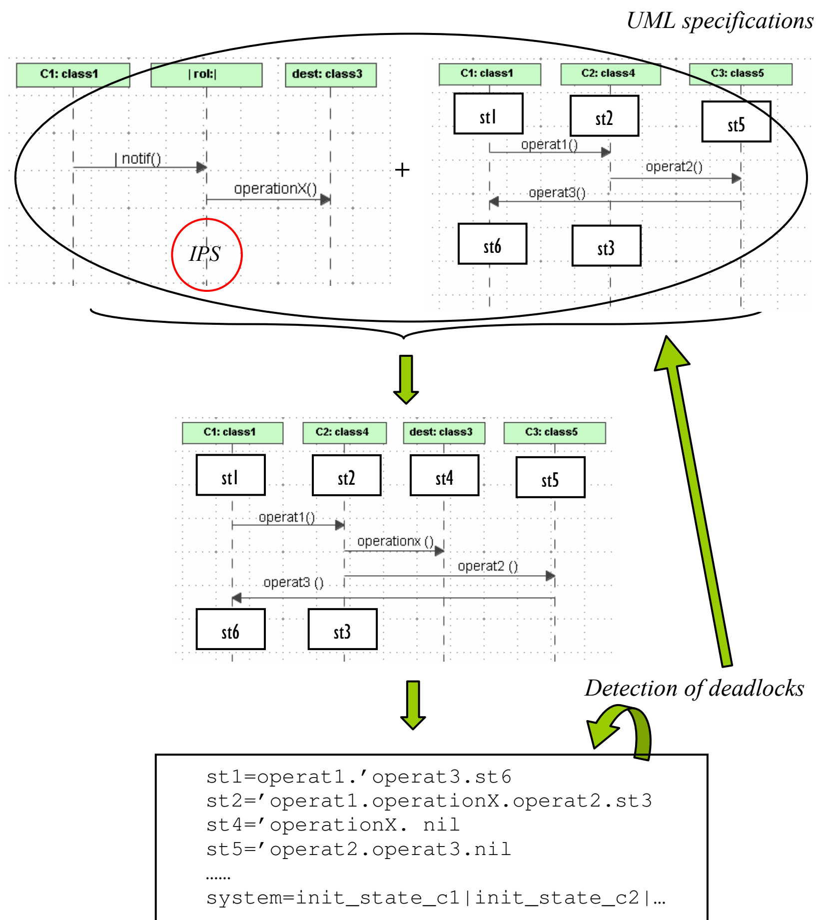

Fig. 2. Instantiation of aspects and refinement of specifications described

Extended state machines are built with validated specifications. The possibility of representing time requirements and complex operations over groups of events (as critical regions) described with UML fragments, and the possibility of achieving a continuous monitoring in the evolution of the state variables of the system are advantages as regards state charts. Each vertex of a machine consists of a structure: <*par, ord, crit, st, variables*> where *par, ord* and *crit* are positive integer variables used for representing parallelism, sequences and critical regions; *st* is a string variable, used for describing the state in which the component is, and *variables* is a set of strings employed for describing the variables used in the conditions and iterations represented on the graph.

Focused on the example, it is possible to build the machines associated to each element of the system. These can be used to simulate the behaviour and to obtain execution traces, useful to prove the built code.

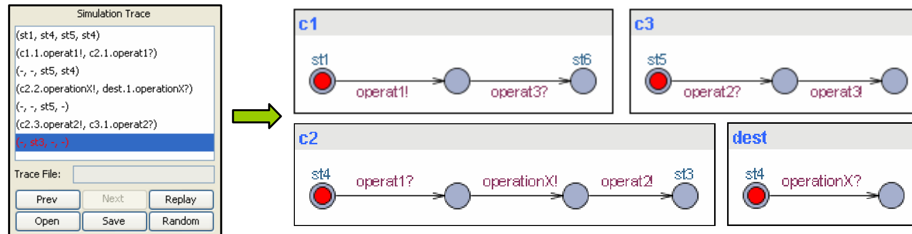

Fig. 3. Obtained machines to apply traces in order to simulate its behaviour

On the other hand, when the system is implemented in Java and AspectJ is used to weave the described aspects, the resulting code can be proved by means of using the sequence of traces obtained from the previous simulation to check if the same results are returned.

Table 1. Aspect code designed for reducing the size of the trace.

```
Public aspect Traceaspect {
      pointcut trace(): execution (c1.*(..))
      ||execution (c2.operationX())
      ||execution (c2.operat1())
      ||execution (c3.operat3());
After(): trace(){
Signature sig = ThisJointPointStaticPart.getSignature();
System.out.println(sig.getDeclaringType().getname()+"."
              + sig.getName());
                    }
  }
```

Another possibility of comparing designed machines with the obtained code consists of creating a new aspect to monitor the execution and build execution traces. Sometimes, these traces can be large and then, there exists the possibility of focusing

the study on a series of events. In order to achieve it, the aspect can be designed in such a way that it only monitors the events of a series of classes and then, state machines whose behaviour is not interesting can be grouped, avoiding references to events that occur inside grouped components (table 1).

This possibility allows us to focus the study on the classes affected by the pointcuts of the aspects of the system. For example, in Figure 4 the result of grouping components 2 and 3, from Figure 3, and the simulation of execution trace obtained from monitor aspect are depicted.

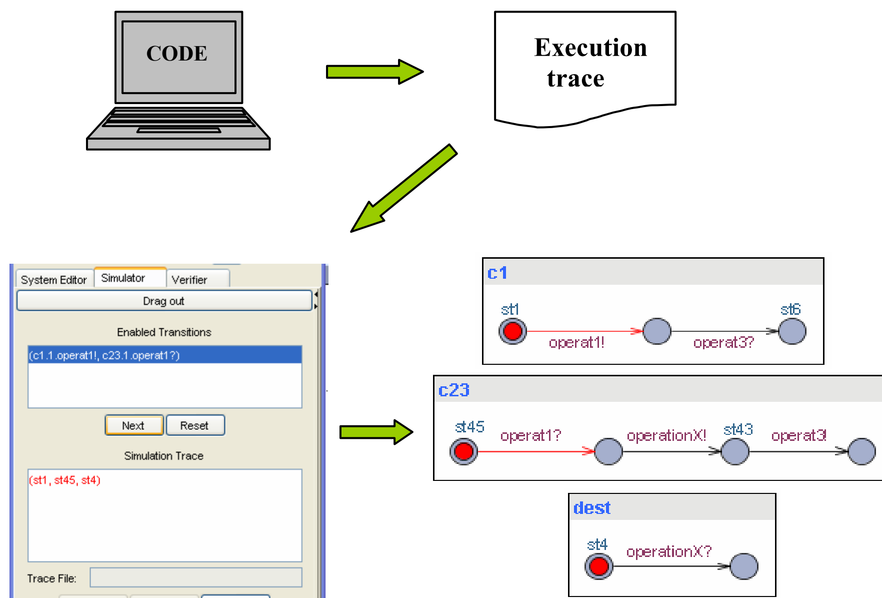

Fig. 4. Simulation of execution trace with c2 and c3 components grouped.

5 Conclusions

AOP facilitates the evolution of a software system. However, the lack of access to the source code of the applications can cause problems in existing software. This paper introduces a practical approximation that evolves the requirements of the system described by means of UML with the built code. These specifications, once validated, will allow us to obtain state machines whose behaviour could be compared with the code of the system, before and after aspects have been included.

In order to make these operations easier, extended state machines are introduced, more complete than traditional statecharts. These machines are adequate to obtain all the possible information from UML sequence diagrams. With these machines, it is possible to study how a software system will be affected when an aspect is included.

It is also possible to perform Model Checking with the properties of the code of the existing system and, finally, the results can be compared with those belonging to the designed machines. Besides, it will be possible to study the resulting woven code when an aspect is included, by means of Model Checking, and trace simulation between state machines and the built code. In order to facilitate these simulations, grouping state machines has been proposed (grouping algorithms is beyond the scope of this paper) to reduce the size of the traces in study and to focus the study on the involved classes.

References

[1] N. McEachen, R.T. Alexander. "Distributing classes with woven concerns: an exploration of potential fault scenarios". Proceedings of the 4th international conference on Aspect-oriented software development.2005, Pages: 192 – 200, ISBN:1-59593-042-6.

[2] S. Katz. "A Survey of Verification and Static Analysis for Aspects". AOSD-Europe-Technion-1. 10 July 2005.

[3] Bandera. http://bandera.projects.cis.ksu.edu

[4] Java Pathfinder. http://javapathfinder.sourceforge.net.

[5] UML homepage. http://www.uml.org

[6] O. Aldawud, T. Elrad, A. Bader. "UML Profile for Aspect-Oriented Software Development",In Proceedings of Third International Workshop on Aspect-Oriented Modeling, March 2003".

[7] M. Mahoney and T. Errad. " Distributing State-Charts to Handle Pervasive Crosscutting Concerns". In Proceeding of Building Software for Pervasive Computing Workshop. OOPSLA 2005.

[8] J. Araujo, J. Whittle, D. Kim, "Modeling and Composing Scenario-Based Requirements with Aspects," re, pp. 58-67, 12th IEEE International Requirements Engineering Conference (RE'04), 2004.

[9] R. B. France, D. Kim, S. Ghosh, E. Song. "A UML-Based Pattern Specification Technique". IEEE Transaction on Software Engineering. March 2004 (Vol. 30, No. 3) pp. 193-206

[10] J. Whittle, J. Schumann. "Generating statechart designs from scenarios". International Conference on Software Engineering. Proceedings of the 22nd international conference on Software engineering. Pages: 314 – 323, 2000, ISBN:1-58113-206-9.

[11] AspectJ. http://www.eclipse.org/aspectj

[12] J. Whitlle, J. Araujo. "Scenario Modeling with Aspects". IEE Proceedings - Software -- August 2004 -- Volume 151, Issue 4, p. 157-171.

[13] S. Uchitel. "Incremental elaboration of scenario-based specifications and behavior models using implied scenarios". ACM Transactions on Software Engineering and Methodology (TOSEM). Volume 13, Issue 1 (January 2004), Pages: 37 – 85, Year of Publication: 2004, ISSN:1049-331X.

[14] 4th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools. in Saint-Louis, Missouri, on 21 May 2005.

[15] J. Whittle, J. Saboo, R. Kwan, "From Scenarios to Code: An Air Traffic Control Case Study," icse, p. 490, 25th International Conference on Software Engineering (ICSE'03), 2003.

[16] UPPAAL. http://uppaal.com

[17] L. Blair, G. Blair. "Composition in Multi-paradigm Specification Techniques". Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS). Pag. 401 – 417, 1999.

[18] M.A. Pérez, A.Navasa, J.M. Murillo, C.Canal. "Definición de máquinas de estados extendidas usadas en descripción de protocolos de interacción". Technical Report TR-23/2006. University of Extremadura (Spain). An English short version can be found in Appendix.

Appendix.

Definition 1. An interaction machine is a graph in the way (V, E, I, Condition, Label, Action, Initial_Vert, Ending_Vert_Set) such that:

- V is a set of vertexes.
- E is the set of edges.
- I is a relation that associates to each edge e ∈ E two vertexes <u,v>∈ V, named the ends, such that u=origin(e) y v=destination(e).
- Condition: E → CondEdge is an injective function that associates one condition to each edge of the graph, where CondEdge is the finite set of condition labels that correspond with the conditions produced in the fragments used for describing the interaction machine of the software element. One condition cond ∈ CondEdge can present null values.
- Label: E → LabelEdges is an injective function for label, where LabelEdges is the finite set of labels that identify the set of messages that can be received or sent, in what concerns the software element that is being described. One label labela ∈ LabelEdges, can present null values.
- Action: E → ActEdges is an injective function that associates one action to each edge of the graph, where ActEdges is the finite set of labels that correspond to the counters produced in the fragments used for describing the interaction protocol of the software element. One action a ∈ ActEdges can present null values.
- Initial_Vert ∈ V is the initial vertex of the graph.
- Ending_Vert_Set ⊂ V is the set of ending vertex of the graph.

Definition 2. Let IM be one interaction machine and let cond ∈ CondEdge be one condition different from empty. Then cond is evaluated as a Boolean and must complete the syntax described in table 2.

Definition 3. Let IM be one interaction machine. Each label labela ∈ LabelEdges different from empty label is composed by a tupla < n, t >, where n is the name of the message (n ∈ N, where N is the finite set of system messages) and t describes the type of event ("!" for representing sending or "?" for representing reception).

Definition 4. Let IM be one interaction machine and let a ∈ Actions be one action different from empty. Then a must complete the syntax described in table 1.

Definition 5. Let IM be one interaction machine and let v ∈ V be a vertex of the graph. Then v is a tupla in the way <par, ord, crit, st, variables> where par, ord y crit are positive integer variables used for representing parallelism, sequences and critical regions; st is an string variable, used for describing the state in which the component

is, and variables is a set of strings used for describing the variables[b] used in the conditions and iterations represented on the graph.

Table 2. Syntax of the conditions and actions of the interaction machine.

```
Expression → ID | NAT
   | Expression ’[’ Expression ’]’| ’(’ Expression ’)’
   | Expression ’++’| ’++’ Expression | Expression ’--’
   | ’--’ Expression | Expression AssignOp Expression
   | UnaryOp Expression | Expression BinaryOp Expression
   | Expression ’.’ ID
UnaryOp  → ’-’ | ’!’ | ’not’
BinaryOp → ’<’ | ’<=’ | ’==’ | ’!=’ | ’>=’ | ’>’ | ’+’ | ’-’
   | ’*’ | ’/’ | ’and’ | ’or’
AssignOp → ’:=’ | ’+=’ | ’-=’ | ’*=’ | ’/=’
```

[b] The variables used in IM must be: parameters from sequence diagrams, internal variables from sequence diagrams, global system variables or clock variables.

Design-Based Pointcuts Robustness Against Software Evolution

Walter Cazzola[1], Sonia Pini[2], and Ancona Massimo[2]

[1] Department of Informatics and Communication,
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it
[2] Department of Informatics and Computer Science
Università degli Studi di Genova, Italy
{pini|ancona}@disi.unige.it

Abstract. Aspect-Oriented Programming (AOP) is a powerful technique to better modularize object-oriented programs by introducing crosscutting concerns in a safe and noninvasive way. Unfortunately, most of the current join point models are too coupled with the application code. This fact harms the evolvability of the program, hinders the concerns selection and reduces the aspect reusability. To overcome this problem is an hot topic.

This work propose a possible solution to the limits of the current aspect-oriented techniques based on modeling the join point selection mechanism at a higher level of abstraction to decoupling base program and aspects.

In this paper, we will present by examples a novel join point model based on design models (e.g., expressed through UML diagrams). Design models provide a high-level view on the application structure and behavior decoupled by base program. A design oriented join point model will render aspect definition more robust against base program evolution, reusable and independent of the base program.

1 Introduction

Aspect-oriented programming (AOP) is a powerful technique to better modularize object-oriented programs by introducing crosscutting concerns in a safe and noninvasive way. Each AOP approach is characterized by a *join point model* (JPM) consisting of the *join point*s, a means of identifying the join points (*pointcut*s) and a means of raising effects at the join points (*advice*). Crosscutting concerns may not be well modularized as aspects without an appropriate join point definition that covers all the interested elements, and a *pointcut definition language* that allows the programmer of selecting them.

Traditionally, the pointcuts allow the programmer of selecting the join points on the basis of the program lexical structure, such as explicit program elements names. The dependency on the program syntax renders fragile the pointcuts definition [2, 11] and strictly couples an aspect to a specific program harming the evolvability [15] and hindering the aspect reusability [7].

At the moment, aspects are not robust against evolutions in the base program. This is because 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 can seriously hinder the software evolution. Thus, this implies that all pointcuts of each aspect need to be checked and possibly revised whenever the base program evolves.

To get the *obliviousness* [3] the aspect programmer should be unaware of the structure and syntax of the base-level program to apply its aspects as well as the base-level programmer must be unaware of the additional aspects. To get a total obliviousness[3] means also to decouple the aspect definitions from the dependency on the structure and syntax of the program they advice, solving the abovementioned problems.

Therefore, the required enhancement should consist of developing a pointcut definition language that supports join points selection on a more semantic way. To provide a more expressive and semantic-oriented selection mechanism means to use a language that captures the base-level program behavior and properties abstracting from the syntactic details. Several attempts in this direction have been done but none of these really approaches the problem in its entireness and in general they raise also new issues, such as efficiency and flexibility. We think that the *design models* provides a more suitable representation to abstract join points identification from the base-code structure and syntax.

In this paper, we propose a design oriented join point model that should offer the right level of abstraction from the base-code. In particular, in our proposal, join points are described by means of UML-like descriptions (basically, activity and sequence diagrams) representing computational patterns, these elements are called *join point patterns*. In other word, we propose of using enriched UML diagrams (or portion of) to describe the control flows or the computational contexts and the join points inside these contexts to detect possible woven points. Pointcuts consist of logic composition of join point patterns. In this way, pointcuts are not tailored on the program syntax and structure but they are more general.

The rest of the paper is organized as follows: in section 2 we overview the limitations against the software evolution of the AspectJ-like join point models, in section 3 we introduce our join point model and in particular the concept of *join point pattern*, finally, in section 4 and in section 5 we face some related works and draw out our conclusions.

2 Limits of the AspectJ-Like JPM against Software Evolution

The join point model has a critical role in the applicability of the aspect-oriented methodology. As stated by Kiczales in his keynote at AOSD 2003 [9] the pointcut definition language has the most relevant role in the success of the aspect-oriented technology.

Most of the AOP approaches use a join point model similar to that of AspectJ [10]. It exploits a dynamic call graph [6] to select the correct join points. The AspectJ pointcut language offers a set of *primitive pointcut designators*, such as **`call`**, **`get`** and **`set`** specifying a method call and the access to an attribute. These primitive pointcut

[3] As *total obliviousness*, we mean the unawareness of the base-level program of the existence of the aspects and vice versa.

designators can be combined using logical operations (||, &&, !) forming more complex pointcuts. All the pointcut designators expect, as an argument, a string specifying a pattern for matching method or field signature. These string patterns introduce a real dependency of the syntax of the base code.

Therefore, most AOP approaches have a tight coupling between aspects and base program, even if the aspect definition is syntactically separated from the base program, changes to the base program can immediately require changes to the aspect definition. Intuitively, since pointcuts capture a set of join points based on some structural or syntactical property, any change to the structure or syntax of the base program could also change the applicability of the pointcuts and the set of captured join points. This is in direct contrast with the general aim of AOP, that is, to make programs easy to read, manage and evolve, by providing new modularization mechanism.

Pointcut heavily relies on how the software is structured at a given moment in time. In fact, the aspect developer subsumes the structure of the base program when he/she defines the pointcuts; the name conventions are an example of this subsumption. The aspect developer implicitly imposes some *design rules* that the base program developer has to follow when evolves his program to be compliant with the existing aspects and avoid of selecting more or less join points than expected. In this case, problems with evolution and obliviousness depend also of the need of guessing these, often silent, conventions.

These rules derive from the fact that pointcuts often express *semantic properties* about the base program in terms of its *structural properties*. For example, the following `setterAccess()` pointcut should capture all the methods that modify the state of the object.

```
pointcut setterMethod() : call(* set*(..));
```

To define this semantic property, the pointcut relies on the coding convention that the name of this kind of methods *always* starts with the prefix `set`. Since the rule subsumed by this pointcut is not imposed by any mechanism, not all developers need to be aware of its existence and, consequently, of having to respect it; in practice this rule gets broken very often. During the base program evolution new methods can be added and existing ones can be removed such that they are captured by the pointcut definition only if they follow the naming convention.

Since, the problem of the evolution in aspect-oriented programs is mainly that the set of join points captured by a pointcut may change when changes are made to the base program, even though the pointcut definition itself remains unaltered. Then, to avoid this problem we need a low coupling of the pointcut definition with the source code.

3 Design-Based Pointcut Language

Design models (UML diagrams, formal techniques and so on) provide the right level of abstraction necessary to have a global and static view of the system and to select the join points thanks to their properties and where they are located (i.e., the context) [2], and then to obtain a more robust pointcut mechanism against the software evolution. We

Pattern-Based Join Point Model: Terminology and Element Description
`Join Points` *They are hooks where code may be added. We consider two different kind of join point, normal join points that represent points of the application behavior where to insert the advice code, and around join points that represent portion of application behavior that must be substituted with the advice code. They are pointed out by one or more join point patterns and refer to the application code.*
`Join Point Patterns` *They are UML diagrams, with a name, that describe a set of join points in terms of their application context. These patterns provide an incomplete and parametric representation of the application behavior. The set of all the declared join point patterns is called the join point pattern space.*
`Join Point Pattern Space` *It is the set of all join point patterns defined into the application.*
`Pointcut` *It is a query on the join point patterns space selecting a set of join points. The queries are created as logic composition of the join point names identified into the join point pattern space.*
`Advice` *It is the code applied at the join points when the associated pointcut is evaluated to true.*

Table 1. Pattern-Based Join Point Model: Terminology and Description

propose to tackle the join point model problems by selecting the join points in terms of the base program design models.

Model-based pointcut definitions are less subject to the *fragile pointcut problem* [11], and then they are more robust against evolution problems, because they are not defined in terms of how the program is structured at a certain point in time. Since, model-based pointcut definitions are decoupled from the structure and syntax of the base program, the fragile pointcut problem is transferred to a more conceptual level. By defining pointcuts in terms of a design model, the fragile pointcut problem has now been translated into the problem of keeping the right localization of the design context and the join points into the base program.

The pointcut definition mechanism we are proposing, called *join point pattern specification language* selects the join points in terms of the base program design models. The application design models provide an abstraction over the application structure. Thanks to this abstraction, the join point patterns can describe the join point position in terms of the application behavior rather than its structure. In other words, we achieve a low coupling of the pointcut definitions with the source code since the join point pattern definition is defined in terms of design model rather than directly referring to the implementation structure of the base program itself.

The join point patterns are graphically specified through a UML-like description — sequence and activity diagrams. A visual approach is more clear and intuitive and makes more evident the separation from the program source code. Finally, UML-like approach is not limited to a specific programming language but can be used in combination with many. At the moment, we are using the `Poseidon4UML` program for depicting the join point patterns but we are developing an ad hoc interface for that.

In general, software evolution involves both structural (e.g., add classes, methods, fields and so on) and behavioral changes, then the pointcuts can affect both the structure and the behavior. In this paper, we only focus on the behavioral join point pattern

```
abstract aspect Observer {
  void notify() { ... }
  abstract pointcut p();
  abstract pointcut c();
  after(): p() {notify();}
  after(): c() {notify();}
}
```

```
aspect Observing1 extends Observer {
  pointcut p(): call(void Buffer.put(int));
  pointcut c(): call(void Buffer.get());
}
```

```
aspect Observing2 extends Observer {
  pointcut p():within(Buffer) && call(* put*(*);
  pointcut c():within(Buffer) && call(* get*(*));
}
```

Fig. 1. The abstract observer pattern aspect with two concrete implementations.

definition; since affecting the application structure simply consists on introducing and removing elements and can be faced as explained in [1].

3.1 The Join Point Pattern Specification Language

In this paper, we borrowed the terms *join point* and *pointcut* from the AspectJ terminology but we use them with a slightly different meaning. The *join point*s are *hooks where code may be added* rather than *well defined points in the execution of a program*. Whereas, the *poitcut*s refer to a set of join points. To complete the picture of the situation, we have introduced a new concept: the *join point pattern* as *a template on the application behavior identifying the join points in their context*. These patterns provide an incomplete and parametric representation of the application behavior. Look at Table1 for a summary and brief description of the elements composing the model.

In addition to decoupling the pointcut definitions from the base code, design-based join point patterns are less fragile to evolution of the base program because the pointcut definitions are based on composition of join points, that are no-linked to the application structure and syntax but linked to the behavior of the application.

A join point pattern is a sample of the computational flow described by using a behavioral/execution flow template. The sample does not completely define the computational flow but only the portions relevant for the selection of the join points. The set of all defined join point patterns is called *join point pattern space*. Each join point pattern can describe and capture many join points; these join points are captured together but separately advised. Pointcuts are expressed as a logic combination of one or more join point patterns.

Now, we will explain the join point pattern definition language "syntax" by examples. Let us consider the implementation of the *observer pattern* [4] as an aspect to observe the state of a *buffer*. The `Buffer` instances originally support only two kinds of operations: to retrieve (`get`) and to insert (`put`) elements in the buffer. The observer will monitor the work of these tow family of methods.

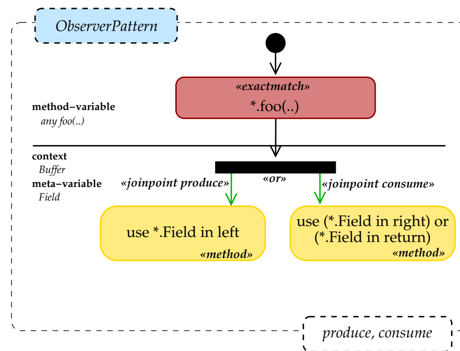

Fig. 2. A Join Point Pattern capturing all the state changes in the `Buffer` class.

Figure 1 shows an abstract aspect (written in AspectJ) that implements the observer pattern behavior with two possible concrete implementation of its pointcuts. The use of an abstract aspect is a way to decouple the crosscut definition from the aspect. The first concrete aspect is based on enumerating the method calls of the base-program, whereas the second one is based on the use of name conventions and wildcards. Both these concrete aspects capture all the interested join points in the case of a buffer implementation which respects the implicit programming conventions proposed from the problem statement, but what happens when the buffer class evolves in a way that violates the self-imposed programming conventions?

To answer to this question, we consider few possible evolutions of the `Buffer` class. First case, we add a method `"void putAll(int [])"` to the `Buffer` class. This event breaks the first concrete aspect because the new method is not listed in the `p()` pointcut. To maintain the expected behavior of the aspect, the pointcut must be modified to include also the new method. The second concrete aspect is more robust and the new method is automatically captured by it because it respects the naming conventions and start by `put`.

Now, let us consider a new change: a method `returnElements` is added. This new method returns a collection with a specified number of elements from the buffer. In this case, both first and second concrete aspect do not capture the join points introduced by calling the new method. The first for the same reason raised in the previous example and the second since the name of the new method does not respect the self-imposed conventions.

Figure. 2 shows a join point pattern capturing *all the method executions which change the state of the* `Buffer` *class*, i.e., our join point pattern can capture both the executions to methods that retrieve data from the buffer and that introduce data in the buffer.

The behavior we are looking for is characterized by: i) the call to a method with any signature, ii) whose body either assign anything to a field of the target object (to select the `put` method family) *or*, either assign a field of the target object to anything or return a field of the target object (to select the `get` method family). This join point

pattern explicitly refers to the concept of a method that change the `Buffer` state rather than trying to capture that concept by relying on implicit rules about the program implementation structure. Consequently, the pointcut defined using this pattern does not need to be verified or changed to be compliant with the evolution of the base program: if the context of the pattern correctly classifies all methods which change `Buffer` state, the pointcut remains correct. By using our `ObserverPattern` the new `putAll(int [])`, and `returnElements(int)` methods will be automatically captured.

The activity diagram describes the context where the join points could be found, more details are used to describe the context and more the join point pattern is coupled to the application code. The use of *meta variables* grants the join point pattern independence from a specific case. In the example, `foo` and `Field` are meta-variables, respectively a *method meta-variable*, i.e., a variable representing a method name and a *variable meta-variable*, i.e., a variable representing a variable name. In this example the method signature is not specified, therefore any method call could be captured if it has the right behavior independently of its signature. If necessary, type meta-variable, i.e., a variable whose values range on types, can be used to define the method signature. Meta-variables got a value during the pointcut evaluation and their values can also be used by the advice.

In the caller swimlane[4], we look for the invocation of the `foo(..)`[5] method whereas in the callee swimlane we look at the method body for either the assignment to a generic class field or, either the use of generic class field into the right of an assignment or the use of the field in a return statement. The former should be an exact statement match, — i.e., we are looking for exactly that call — whereas in the latter we are looking for a specific use of a field in the whole method body. This difference can be expressed by using the join point pattern syntax and a couple of stereotypes:

– a rounded rectangle, called *template action*, indicates that we are looking for the use of a meta-variable in the next statements, a stereotype set a constraint for the searching scope; ≪`method`≫ limits to the method body whereas ≪`block`≫ limits to the current block;
– we can look for the use of a meta-variable in a left (`left`) or right (`right`) part of an assignment, in a boolean expression (`booleanCondition`), in a generic statement (`statement`), and in a return statement or in their logic combination;
– a rounded rectangle with the ≪`exactmatch`≫ stereotype, called (according to UML) *action*, indicates one or more instructions, expressed following the Java syntax; the names used inside this block can be either meta-variables, constant variable names or if not useful to the pattern definition indicated as `(i)` with $i \in \mathbb{N}$.

The join point possible location is indicated by the ≪`joinpoint`≫ stereotype attached to an arrow. Each join point pattern can describe the context for many join points that can be located by using a ≪`joinpoint`≫ stereotype with a different name. All the captured join points are listed in the window in the low-right corner of the join point pattern specification. In Fig. 2 we have two different join points called respectively `produce` and `consume`.

[4] A swimlane is a way to group activities performed by the same actor/object.
[5] Please note that `foo(..)` is meta-variable and method signature is not specified.

We have adopted a loose approach to the description of the computational flow. In the join point pattern, based on activity diagrams, the lines with a solid arrowhead connecting two elements express that the first one follows immediately the other, and the lines with a stick arrowhead (see Fig.2) express that the first follows the other, but not immediately, i.e., zero or more *not relevant* actions[6] could happen before the second action, the number of actions that could happen is limited by the scope.

Our join point model is strictly based on the structure of the computational flow, so we don't need to differentiate between `before` and **`after`** advice but we can simply attach the ≪`joinpoint`≫ stereotype in the right position, i.e., before or after the point we would like to advice. A special case is represented by the *around join point pattern*s which match portions of the behavior instead of a single point; the whole matched portion represents the join point and will be substituted by the advice code.

3.2 Aspects that Use Join Point Patterns

The showed join point pattern simply describes where the join points can be found, to complete the process we must declare an aspect where the join point pattern is used to associate the advice code at the interested join points.

The aspect definition, like in most AOP languages, includes pointcuts definition and advices linked to these pointcuts. Moreover, the aspect must declare all the join point patterns it uses and which join points it imports from them. Both pointcuts and advices will use these information in their definition.

The following `Observer` aspect imports the `produce` and the `consume` join points from the `ObserverPattern` join point pattern.

```
public aspect Observer {
    void notify() { ... }
    public joinpointpattern ObserverPattern(produce, consume);
    public pointcut p(): produce();
    public pointcut c(): consume();
    advice() : p() {notify();}
    advice() : c() {notify();}
}
```

4 Related Works

This paper propose to decoupling base programs and aspects using an UML-based join point model by approaching the join points selection on a less syntactical and structural basis. To get a semantic join point model to avoid the fragile pointcut problem is a quite hot topic and several approaches are currently under investigations.

In [13], Noguera et al. present a mechanism to express type-safe source code templates in pure Java that improves the expressiveness of pointcut languages. To have a more semantic pointcut language, they propose to match, not only on the signature, but

[6] 3These actions do not participate in the description of the join point position, so they are considered not relevant.

also on the structure of the method. They propose a way to extend AspectJ pointcut language with structural constructs in the form of typesafe native Java source code templates, where templates, define a source code model in which some elements are variable. The basic idea is similar, i.e., identify join points not only on the base of method signature but also on method behavior.

In [7] Kellens et al. propose a novel technique of model-based pointcuts, which translates the fragile pointcut problem to a more conceptual level where it is easier to solve. This is done 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.

In [5] Gybels et al. present a logic-based crosscut language, called CARMA. The use of a crosscut language based on logic programming it gets the use of unification as a more advanced wildcard mechanism, the use of logic rules for writing reusable pointcut.

In [8] Kellens et al. present a method for keeping the conceptual model documentation consistent with the source code when the program evolves. In particular they implement a particular solution to the fragile pointcut problem through an extension of the CARMA aspect language combined whit the formalism of intensional views [12].

Pointcut delta analysis [14] tackles the fragile pointcut problem by analyzing the difference in captured join points, for each pointcut definition, before and after an evolution. Their approach to deal with the fragile pointcut problem for current languages.

Although such expressive pointcut languages permit to render pointcut definitions much less fragile, but none of these languages approaches the problem in its entireness. A pointcut definition still needs to refer to specific base program structure or behavior to specify its join points. This dependency on the base program remains an important source of fragility.

5 Conclusions

Current AOP approaches suffer from well known problems that rely on the syntactic coupling established between the application and the aspects. This is a serious inhibitor to evolution of aspect-oriented programs. A common attempt to give a solution consists of freeing the pointcut definition language from these limitations by describing the join points in a more semantic way.

This paper shows the robustness against evolution of a design-based approach to join points identification. This approach allows of decoupling aspects definition and base-code syntax and structure, and of rendering the pointcut definitions less fragile against the base program evolution. Pointcuts are specified using UML-based join point pattern. More precisely, a join point pattern is a template on the application behavior identifying the join points in their context. In particular join points are captured when the pattern matches portion of the application behavior.

Compared with current approaches, we can observe some advantages; first of all, we have a pointcuts definition more behavioral. In the join point pattern definition we identify the context of the computational flow we want to match, and precise point we want to capture, weaken the coupling of the aspect to the base program and hence,

providing crosscuts that are more robust towards evolution. The graphical definition of join point patterns is more intuitively and comprehensible for programmers. Moreover, a visual view of the context in which locate the join points would be preferred since it better demonstrates where and how an aspect can influence a program.

References

1. Walter Cazzola, Antonio Cicchetti, and Alfonso Pierantonio. Towards a Model-Driven Join Point Model. In *Proceedings of the 11th Annual ACM Symposium on Applied Computing (SAC'06)*, pages 1306–1307, Dijon, France, on 23rd-27th of April 2006. ACM Press.
2. Walter Cazzola, Jean-Marc Jézéquel, and Awais Rashid. Semantic Join Point Models: Motivations, Notions and Requirements. In *Proceedings of the Software Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT'06)*, Bonn, Germany, on 21st March 2006.
3. Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, USA, October 2000.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.
5. Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 60–69, Boston, Massachusetts, April 2003.
6. Erik Hilsdale and Jim Hugunin. Advice Weaving in AspectJ. In *Proceedings of the 3rd Int'l Conf. on Aspect-Oriented Software Development (AOSD'04)*, pages 26–35, Lancaster, UK, March 2004.
7. Andy Kellens, Kris Gybels, Johan Brichau, and Kim Mens. A Model-driven Pointcut Language for More Robust Pointcuts. In *Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT'06)*, Bonn, Germany, March 2006.
8. Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*, Nantes, France, July 2006. Springer.
9. Gregor Kiczales. The Fun Has Just Begun. Keynote AOSD 2003, Boston, March 2003.
10. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeff Palm, and Bill Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327–353, Budapest, Hungary, June 2001. ACM Press.
11. Christian Koppen and Maximilian Störzer. PCDiff: Attacking the Fragile Pointcut Problem. In *Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS'04)*, Berlin, Germany, September 2004.
12. Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving Code and Design Using Intensional Views - A Case Study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, July/October 2006.
13. Carlos Noguera and Renaud Pauwlak. Open Static Pointcuts Through Source Code Templates. In *Proceedings of Open and Dynamic Aspect Languages Workshop (ODAL'06)*, Bonn, Germany, March 2006.
14. Maximilian Störzer and Jürgen Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656, Budapest, Hungary, September 2005.

15. Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Chai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. On the Criteria to be Used in Decomposing Systems into Aspects. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, Lisbon, Portugal, September 2005.

Tools and Middleware for Software Evolution

Chairman: Mario Südholt, École des Mines de Nantes

Evolution of an Adaptive Middleware Exploiting Architectural Reflection

Francesca Arcelli and Claudia Raibulet

DISCo – Dipartimento di Informatica Sistemistica e Comunicazione,
Università degli Studi di Milano-Bicocca,
Via Bicocca degli Arcimboldi, 8, 20126, Milan, Italy
{arcelli, raibulet}@disco.unimib.it

Abstract. Nowadays information systems are required to adapt themselves dynamically to the ever changing environment and requirements. Architectural reflection represents a principled means to address adaptivity. It also represents an emerging approach to deal with the software evolution issues. In this paper we aim to point out how systems exploiting architectural reflection to achieve adaptivity evolve in an organized, linear manner controlling easier their growth and complexity than systems based on ad hoc solutions. To sustain this affirmation we present the possible evolution improvements we gain through our Adaptive and Reflective Middleware (ARM).

Keywords: Software evolution, architectural reflection, adaptive systems.

1 Introduction

One of the most challenging issues raised by nowadays information systems is to adapt themselves dynamically and automatically in the attempt to accomplish the anytime, anyone, anywhere paradigm in a constantly changing reality. In this context, we describe our approach with the aim to provide support to identify, choose, and exploit the appropriate system's components able to satisfy users' requests according to different levels of quality of services in a dynamic mobile-enabled heterogeneous environment.

Our solution for adaptive systems exploits reflection [13] at the architectural level. Reflection (or computational reflection) has become popular because of the mechanisms it provides to a system to observe and control itself by using appropriate metadata. Initially, reflection has been successfully used by the programming language community. Today, its benefits are extended to the architectural level. Architectural reflection [2, 5] introduces additional layers playing an intermediary role between the representation and implementation of the system's components/functionalities, and applications. These reflective layers enable applications to adapt to the systems' features and, vice-versa, systems to adapt to the applications' requirements. Several works [1, 5, 22] aim to define the main elements of the architectural reflection: *base* objects or levels, which define the components of the system to be inspected, *meta* objects or levels, representing a reification of the

base objects/levels, and the *causal connection* mechanism [13], which enables the synchronization between base and meta objects/levels.

The usage of reflection at the architectural level has both advantages (i.e., a principled, as opposed to ad-hoc, way to achieve adaptivity [8], an explicit representation of architectural aspects exploited at run-time) and disadvantages (i.e., a significant increase of the number of software components which may reduce overall efficiency, modifications of the reflective components may cause overall damage). In this paper we focus on an additional and implicit advantage, which can be considered a side-effect of applying reflection at the architectural level: the support for software evolution [4, 16]. The benefits of using architectural reflection as a mechanism to achieve software evolution are treated in several scientific works [3, 7, 9, 14, 20, 21].

Our aim is to point out those design aspects of a reflective architecture (or more precisely of the reflective knowledge and its management) which ensure implicitly its proper and consistent evolution. To achieve this goal, we present our solution for a reflective and adaptive middleware (ARM) considering the main laws of software evolution introduced by Lehman [11]: continuous adaptation, increasing complexity, continuing growth, declining quality, organizational stability, and conservation of familiarity.

The rest of the paper is organized as follows. Section 2 provides a brief overview of our software architecture exploiting reflection to achieve adaptivity. Section 3 focuses on the design aspects of our approach that enable and ensure the evolution of ARM considering the Lehman's software evolution laws. Related works are described within Section 4. Conclusions and further work are dealt within Section 5.

2 ARM: An Adaptive and Reflective Middleware

Our solution to achieve runtime adaptivity in service-oriented information systems [10] defines a reflective middleware composed of two layers (see Figure 1). The first layer defines the reflective knowledge. It reifies the system's components in terms of *reflective objects* (capturing their current state) and their related *quality of services* (QoS) [6, 17]. For example, we represent a display device with its dimension, resolution, color depth, or a transmission device with its bandwidth and latency. In addition, we have introduced the concept of *property*. Properties [18] express the characteristics of the system's components that are not directly measurable at run-time, but which are exploited together with the QoS to achieve adaptivity. Currently, ARM defines three properties: *structural*, representing the physical structure of an ARM-enabled node, *topological*, representing the connections between ARM-enabled nodes, and *location*, representing the physical location of a component.

The second layer introduces the concept of *views* [18] on the reflective knowledge, which represent organizational mechanisms of the reflective entities based on various criteria. Views organize reflective objects based on their QoS, structure, location, and topology. Each view has associated strategies that implement the logic necessary to take decisions. For example, strategies identify the most appropriate system's component(s) to execute a service based on its QoS or on its location. Strategies

depend strongly on the application domain, thus they have not been inserted into views, but represented through a separate class and associated to views.

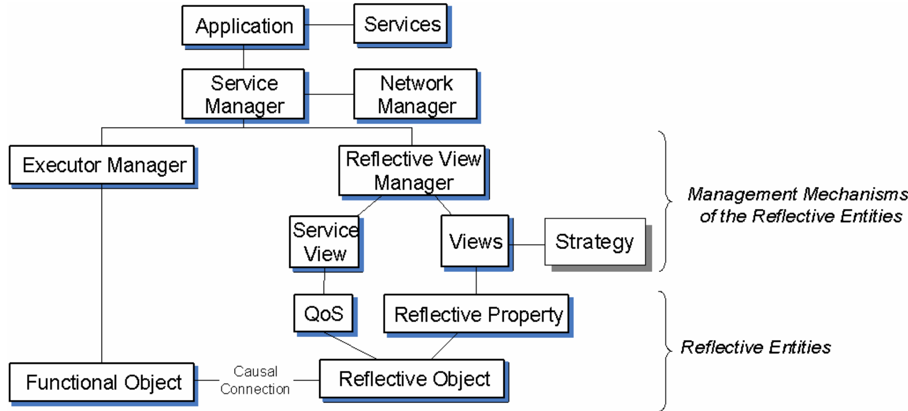

Fig. 1. An Adaptive and Reflective Architecture

To explain how our architecture exploits its reflective elements to achieve adaptivity we describe how it chooses the most appropriate component to execute a service characterized by specific QoS and properties. Users' requests of services and their related QoS/properties arrive to the service manager, which may interrogate both the local ARM node and/or the remote ARM-enabled nodes. Once the service manager identifies the type of request (i.e., service execution, inspection of the available services, etc.), the request arrives to the reflective view manager. Based on the service required and its QoS/properties, the reflective view manager determines how to organize better the reflective knowledge to search for the most appropriate component that may fulfill the current service request. It creates dynamically views on the reflective knowledge to address efficiently a request. Views identify the most appropriate component to execute a service based on their own semantic. By composing the partial results provided by each view, we obtain the best component available on the local ARM-enabled node, which is further compared with the remote results provided by the network manager from remote ARM-enabled nodes in order to obtain the best overall solution. The references of the identified components are then passed by the service manager to the executor manager to inform the last which component should execute the service.

Eventual modifications performed on the identified reflective entities for the execution of the requested services are propagated to the functional objects through a causal connection mechanism. Finally, the executor manager executes the service exploiting the components identified by the reflective view manager.

2.1 The Functional vs. the Reflective Part of the Architecture

In our solution, the architecture has two main parts: the *functional* part, which defines and implements the functionalities of a system, and the *reflective* part, which defines the knowledge and the mechanisms to achieve adaptivity. The two parts have only one connection point: the causal connection between the functional and reflective objects. The objective of the causal connection is to ensure the synchronization between the two architectural parts: a modification of a functional object is automatically propagated to its correspondent reflective object, and vice-versa, a modification of a reflective object is automatically propagated to its corresponding functional object. Note that not all the functional objects are reified at the reflective layers, but only those who are meaningful at run-time to achieve adaptivity.

Based on the service request, the reflective part of the architecture may or may not be exploited. When adaptivity is required, the reflective part identifies the most appropriate available resources to execute the current service request.

We have considered three main types of request:

- *non adaptive* requests, in which the reflective part of the system is not exploited; requests are forwarded by the service manager directly to the executor manager; an example of such a request is: "print this document on the hp370 printer";

- *low-adaptive* requests, in which the reflective part of the system is exploited to set the QoS of a specified device on the desired values; for example, "print this document on the hp370 printer, in an A3 format, colored, and with a maximum resolution";

- *high-adaptive* requests, in which the reflective part of the system is exploited to choose the most appropriate device to execute the service and, if necessary, to set its QoS as close as possible to the required once; for example, print this document on the nearest printer, in an A3 format, colored, and with a maximum resolution.

Maintaining separately the functional and the adaptive parts, we may modify the reflective entities or their management mechanisms without causing modifications on the functional part; or, the adaptive part is not affected when the functional one (its current implementation) is modified. Usually, adaptivity is implemented at the application level (also because it is strongly domain dependent – for example, strategies which include the decision logic are implemented by applications), fact that makes it hardly reusable or extensible. Our solution defines the main mechanisms necessary to achieve adaptivity, mechanisms which are highly reusable, extensible and/or customizable.

3 Evolution of ARM-Based Systems Exploiting Architectural Reflection

It is well known that supporting the activities involved in software evolution is a very expensive task. Hence, developing software architectures or environments which enable the development of software easier to change, extend and adapt to new requirements or contexts is certainly of great relevance for software evolution and maintenance.

Continuous adaptation is one the well known Lehman's laws of software evolution [11] and certainly one of the main aims of our software middleware for adaptive information systems, where adaptation can occur at different levels and at different steps during the development lifecycle. In the following, we focus on the advantages provided by our approach from the software evolution point of view by considering the main Lehman's laws.

Continuous adaptation refers to the ability of a system to address new requirements and changes. In ARM, evolution aspects may regard the representation and/or the management of the reflective knowledge. The representation can be easily extended or changed because reflective objects capture only the state of a system component, while QoS and properties are modeled as separate entities. Furthermore, QoS and properties can be modified, added or removed independently because they depend only on the underlying system's components and features.

The management of the reflective knowledge is performed through two mechanisms: views and strategies. Views organize reflective knowledge based on various semantics, each one capturing an independent and orthogonal aspect of the reflective entities. Each view has its own strategies, which implement the policies to choose the most appropriate component to execute a service. Views and strategies can be modified, added or removed independently of other software entities in the system.

In addition, the causal connection between the base and meta levels ensures the consistency between the functional, and the reflective, the adaptive part of the architecture.

Separation of concerns [13], the fundamental requirement of reflection, is achieved: reflective knowledge (Reflective Objects) is separated from the base knowledge (Functional Objects). The reflective layers provide only non-functional information about the system being causally connected with the physical layer which provides its functional information. In this way, overall change is avoided due to the fact that modifications within the reflective layers cannot change the functionalities of a system, it can only influence its performances

Increasing complexity as a consequence of the system evolution states that changes in a system lead to the modification of its structure and, implicitly, to an increase of its complexity.

We assert that the complexity of our architecture does not change during its evolution. Its skeleton is composed of five main elements: reflective entities, QoS, properties, views, and strategies. A modification of the reflective and adaptive part of the system regards one or more of these elements. They are modeled independently by separate entities, hence modifications are made separately on each type of element. Their changes cannot increase the overall complexity. In our approach, reflective knowledge is managed through strategies hence, modifications at the reflective layers should not cause modifications at the application layer. The addition of further QoS or properties, or views or strategies maintains complexity unchanged. For example, strategies are implemented exploiting the Strategy design pattern [12], introducing a new strategy in the system means the addition of a new object. In our case evolution is translated into an increase of the number of objects, and not in an increase of the overall complexity.

Continuing growth regards the continuously increase of the functionality offered by a system to maintain user satisfaction.

The functionality of the reflective and adaptive part of our architecture is to identify the proper system's components to satisfy service requests. Due to the fact that the reflective knowledge is causally connected to the base entities, when additional services are added to the system they are reified at the meta-level, too. To improve the functionality of the reflective part, various properties and views can be added, for example, a property that specifies the cost of a service or its provider, hence we can have a view on the reflective knowledge based on the newly added property.

The continuing growth does not lead to a *declining quality*, because reflective layers maintain their primary structure. To maintain or improve the quality of the reflective layers, old properties and views can be replaced by new once. New organizations of the reflective objects do not lead to a re-engineering of the reflective layers, the main mechanisms of the representation and management of the reflective knowledge remain unchanged. This ensures implicitly both the *organizational stability* and the *conservation of familiarity* with the reflective layers.

3.1 ARM's Design Issues Improving Evolution

In this section we introduce further aspects which contribute to the evolution and maintenance of ARM.

As previously mentioned, views organize reflective entities based on various semantics according to QoS and properties. From the evolution point of view they provide at least two main advantages: the possibility to extend the number of views on the reflective objects, implicitly to improve the system's adaptivity, and to represent and exploit information such as location, costs, topology, providers in the adaptivity process together with the QoS of the reflective entities. Note that a reflective entity may be used in various views, but it has only one representation in the system, each view containing a list of references to the entities it manages. In this way consistency among views is implicitly achieved. Based on its semantic, a view associates a score to its reflective entities. The reflective entity with the highest score represents the most appropriate one to provide the service claimed by the current request.

To further improve its evolution and maintenance several design patterns [12] have been applied:

- *chain of responsibility pattern* to implement the service view; it addresses two main problems: the dynamic control of a collection of service views, and the management of complex services (i.e., a *send e-mail service* may be seen as a composition of two elementary services *type* and *send e-mail*);
- *composite pattern* to implement the structure of strategies of the service view; the strategy analyzing a complex service is a composition of strategies related to the elementary sub-services of the complex one; this improves significantly the implementation of strategies by requiring the definition of the elementary strategies, and the definition of the complex once as the composition of the already defined strategies; modifications of elementary strategies are automatically propagated to the complex once;

- *strategy pattern* to implement the policies based on which views assign scores and choose the most appropriate entity for a service request;
- *observer pattern* to implement the causal connection mechanism; this pattern provides an efficient mechanism for the synchronization between the base and the reflective objects.

Trying to accomplish a well-defined delimitation of the various aspects of an adaptive and reflective architecture/system, we implicitly achieve maintainability, reusability and integrability.

4 Related Work

Several works address evolution through techniques coming from the software architecture community, especially architectural reflection. Therefore, reflection at the architectural level is exploited more and more frequently to enable and improve software evolution. The main difference between our solution and the other works is that, in ARM, software evolution is an implicit advantage, a side-effect of applying architectural reflection to achieve run-time adaptivity.

In the following we consider only three of the most relevant related works [1, 3, 19] on using reflection for software evolution focusing on the similarities and differences with our approach.

In [1], authors assert that computational reflection [13] provides a programming mechanism which enhances extensibility, reuse and maintenance of a software system. Thus, they define a run-time environment focusing on the main elements that should be defined when applying reflection. Even if the paper does not deal with the advantages of using reflection at the architectural level to enable software evolution, we consider it relevant in that it points out how reflection improves extensibility and maintenance also at the computational level.

[3] describes how reflection may be used at the architectural level to achieve the evolution of a system, which has to face run-time changes. In this case, the objective of the meta-level is to supervise the evolution of the underlying system. To achieve its goal, the meta-level defines two main elements: an evolutionary meta-object, which plans the possible evolutions of the system to satisfy the adaptation requests, and a consistency checker object, which validates the solutions proposed by the evolutionary object. The main similarity between our solution and the one proposed in [3] is that both consider the adaptation of a system to the run-time changes. The main difference (which is very subtle) is the primary objective of using reflection. ARM aims to achieve adaptivity (which implies also evolution); it defines the reflective knowledge and its related management mechanisms in such a way to efficiently choose the most appropriate solution for the current request. The solution may or may not lead to changes in the system, hence to its evolution. The reflective layer proposed in [3] aims to ensure the evolution of a system against run-time changes. Thus, the reflective mechanisms are designed from another point of view although having the same final objective: adaptation to changes.

The approach presented in [20] starts from the idea that the development of a software system involves the manipulation of several views on the system and the coupling of these views is fundamental in achieving the system's evolution. Therefore, in [20], architectural reflection represents the glue between the low-level aspects of a system (described at the base level, i.e., source code) and its high-level aspects (described at the reflective level: design, deployment, interaction). Through its mechanisms which are defined to synchronize the various views, reflection ensures an automatic identification of the inconstancies between the low and high level views enabling an easier evolution of a system. There is a main similarity between our solution and the one proposed in [20]: both introduce the concept of views. However, in ARM, views have been defined for an efficient management of the reflective knowledge: views consider knowledge from different angles enriching and diversifying in the same time the perspectives on the information related to a software system. While in [20], views are associated to the different phases and development aspects of a system. The author defines a use case, a class, a deployment, an interaction and a code view. He aims to have a single and common representation of a system on which to look at from various points of view. Implicitly, the common representation ensures the consistency among views.

5 Current and Further Work

In this paper we briefly described how adaptive systems developed through the ARM approach and hence, exploiting architectural reflection, are easier to maintain. The evolution of these systems, as the capacity to adapt themselves to environment and requirements changes, is largely improved. Representing explicitly information (usually non-functional) necessary to achieve adaptivity makes them easier to understand, maintain and evolve. Hence, through our approach we aim to provide a way to preserve the quality of an adaptive software system independently from its size and complexity.

Reflection is a key feature for *architecture centered evolution:* all the architectural relevant changes made at the architectural level have to be reflected at the code level, assuring synchronization between the two levels. Often software evolution is focused mainly on software maintenance and defect repairs, where only source code evolves, and not the architecture and design. In our approach we adopt an architecture-centered development process and evolution.

Further work will focus on other important aspects such as resource negotiation and allocation as well as mechanisms to choose dynamically strategies based on the application domain. We would like also to explore, as outlined in [14], how reflective modelling of software architectures can support run-time adaptive software evolution. Future work on ARM includes also its evolution towards a service-oriented architecture. This implies the extension of the current architecture in order to consider both the services provided by the underlying hardware components, as well as the services provided by software applications [19].

ARM has been developed by extending and evolving the software architecture designed during the MAIS (Multichannel Adaptive Information Systems) project

[15]. We have reused and improved the representation of the reflective objects and their related QoS, as well as the causal connection mechanism. Furthermore, we have significantly improved the management of the reflective knowledge by introducing properties and views on the reflective entities. Strategies, also present in MAIS [2] and associated to the reflective objects (one specific strategy for each reflective object), have been redesigned and now they can be reused in various views and application domains from multimedia to telemedicine, or from video-surveillance to disaster recovery applications [18].

References

1. Ancona, M., Cazzola, W.: The Essence of Reflection: a Reflective Run-Time Environment. In Proceedings of the 2004 ACM Symposium on Applied Computing, ACM Press, Cyprus, (2004) 1503-1507
2. Arcelli, F., Raibulet, C., Tisato, F., Adorni, M.: Architectural Reflection in Adaptive Systems. In Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), Banff, Alberta, Canada, June, (2004) 74-79
3. Cazzola, W., Ghoneim, A., Saake, G.: Software Evolution through Dynamic Adaptation of Its OO Design. In Objects, Agents and Features, Lecture Notes in Computer Science, Vol. 2975. Springer-Verlag, (2004) 67-80
4. Cazzola, W., Pini, S., Ancona, M.: The Role of Design Information in Software Evolution. In Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'2005). Glasgow, Scotland. (2005) 59-72
5. Cazzola, W., Sosio, A., Savigni, A., Tisato, F.: Architectural Reflection. Realising Software Architectures via Reflective Activities. In Proceedings of the 2nd International Workshop on Engineering Distributed Objects. Lecture Notes in Computer Science, Springer-Verlag (2000) 102-115
6. Chalmers, D., Sloman, M.: A Survey of Quality of Service in Mobile Computing Environments. IEEE Communications Surveys. (1999) 2-10
7. Dowling, J., Cahill, V.: Dynamic Software Evolution and the k-Component Model. In Proceedings of OOPSLA 2001 Workshop on Software Evolution. (2001)
8. Elianssen, F., Andersen, A., Blair, G.S., Costa, F., Coulson, G., Goebel, V., Hansen, O., Kristensen, T., Plagemann, T., Rafaelsen, H. O., Saikoski, K. B., and Weihai, Yu.: Next Generation Middleware: Requirements, Architecture, and Prototypes. In Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'99). (1999) 60-65
9. Ebraert, P., Tourwe, T.: A Reflective Approach to Dynamic Software Evolution. In Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution. (2004) 37-44
10. Erl, T. Service-Oriented Architecture: Concepts, Technology and Design, Prentice Hall PTR, USA, 2005
11. Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., Turski, W.M.: Metrics and Laws of Software Evolution – The Nineties Views. In Proceedings of the 4th International Symposium on Software Metrics, IEEE CS Press. (1997) 20-32
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading MA, USA, (1994)
13. Maes, P.: Concepts and Experiments in Computational Reflection. In Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87). (1987) 147-155

14. Masuhara, H., Yonezawa, A.: A Reflective Approach to Support Software Evolution. In Proceedings of International Workshop on the Principles of Software Evolution. (1998) 135-139
15. MAIS Project – www.mais-project.it
16. Mens, T.: Challenges in Software Evolution. In Proceedings of the International ERCIM-ESF Workshop on Challenges in Software Evolution (ChaSE). Berne Switzerland, (2005)
17. OMG Adopted Specification. UML Profile for Modelling Quality of Service and Fault Tolerance Characteristics and Mechanisms. ptc/2004-06-01, http://www. omg.org, 2004
18. Raibulet, C., Arcelli, F., Mussino, S., Riva, M., Tisato, F., Ubezio, L.: Components in an Adaptive and QoS-based Architecture. In Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'06), Shanghai, China. (2006) 65-71
19. Raibulet, C., Arcelli, F., Mussino, S.: Exploiting Reflection to Design and Manage Services for an Adaptive Resource Management System. In Proceedings of the International Conference on Service Systems and Service Management (IC SSSM'06), IEEE CS Press, Troyes, France. (2006)
20. Rank, S.: Architectural Reflection for Software Evolution. In Proceedings of the 2nd Workshop on Reflection, AOP and Meta-Data for Software Evolution. (2005)
21. Rank, S.: A Reflective Architecture to Support Dynamic Software Evolution. PhD Thesis, Department of Computer Science, University of Durham, UK (2002)
22. Suzuki, J., Yamamoto, Y.: OpenWebServer: An Adaptive Web Server Using Software Patterns. IEEE Communications Magazine, Vol. 37, No. 4, (1999) 46-52

An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution*

Javier Camara[1], Carlos Canal[1], Javier Cubo[1], Juan Manuel Murillo[2]

[1]Dept. of Computer Science, University of Málaga (Spain)
{jcamara,canal,cubo}@lcc.uma.es
[2] University of Extremadura (Spain),
[2]Dept. of Computer Science, Quercus Software Engineering Group,
juanmamu@unex.es

Abstract. This paper briefly describes the design of a dynamic adaptation management framework exploiting the concepts provided by Aspect-Oriented Software Development (AOSD) -in particular Aspect-Oriented Programming (AOP)-, as well as reflection and adaptation techniques in order to support and speed up the process of dynamic component evolution by tackling issues related to signature and protocol interoperability. This will provide a first stage to a semi-automatic approach for syntactical and behavioural adaptation.

1 Introduction

One of the most significant trends in the software development area is that of building systems incorporating pre-existing software components, commonly denominated commercial-off-the-shelf (COTS). These are stand-alone products which offer specific functionality needed by larger systems into which they are incorporated. The purpose of using COTS is to lower overall development costs, reducing development time by taking advantage of existing and well tested products. But this approach to systems engineering has its drawbacks: development teams have no control over the functionality, performance, and evolution of COTS products because of their Black-Box nature. Moreover, in most of the cases these components are not designed to interoperate with each other, requiring customized adaptation which has to be performed time and again when teams face their integration along the evolution of the system. These activities are highly demanding, consuming time and resources which could otherwise be devoted to the enhancement or development of new functionality.

The need to automate the aforementioned adaptation tasks has driven the development of Software Adaptation (SA) [5], a field characterized by highly dynamic run-time procedures that occur as devices and applications move from network to network, modifying or extending their behaviour. SA promotes the use of software adaptors [22], specific computational entities for solving interoperability

* This work has been supported by Spanish MCYT Project TIN2004-07943-C04-01.

problems between software entities (i.e. components) which can be classified in four different levels:

Signature Level: Interface descriptions at this level specify the methods or services that an entity either offers or requires. These interfaces provide names, type of arguments and return values, or exception types. This kind of adaptation implies solving syntactical differences such as method names, argument ordering and data conversion and synthesis.

Protocol Level: Interfaces at this level specify the protocol describing the interactive behaviour that a component follows, and also the behaviour that it expects from its environment. Indeed, mismatch may also occur at this protocol level, because of the ordering of exchanged messages and of blocking conditions. The kind of problems that we can address at this level is, for instance, compatibility of behaviour, that is, whether the components may deadlock or not when combined.

Service Level: This level groups other sources of mismatch related with non-functional properties like temporal requirements, security, etc.

Semantic Level: This level describes what the component actually does. Even if two components present perfectly matching signature interfaces, they also follow compatible protocols, and are compatible at the service level as well, we have to ensure that the components are going to behave as expected.

This work is focused in the design of a framework based on Software Adaptation techniques and how these can be applied in order to support and speed up the process of Software Evolution, particularly at the signature and protocol levels. Considering the aforementioned opaque nature of COTS components, the techniques provided for the development of this framework should be non-intrusive. In this sense, AOP [10] makes a perfect candidate, providing mechanisms to extend and modify the behaviour of components without directly altering them (i.e., their code). Automatic and dynamic procedures are also required in order to enable adaptation just in the moment in which components join the context of the system (or are substituted as the system is running). The development of this kind of framework can provide a new breeding ground for the development of agile methodologies for Software Evolution by reducing integration effort through the support of (semi)automatic component adaptation.

In this paper, Section 2 discusses the advantages provided by different approaches to dynamic AO component adaptation, and justifies the convenience of selecting Dynamic Adaptor Management. Although signature level is the state-of-the-art in adaptation (e.g. CORBA's IDL-based signature description), several proposals have been made in order to enhance component interfaces with a description of their concurrent behaviour [2, 4, 12], allowing automatic adaptor derivation in some circumstances [3]. Section 3 briefly describes the design of a dynamic adaptation management framework based on the concept of automatic adaptor derivation and gives some tips on implementation issues using AspectJ. At last, section 4 presents some conclusions and open issues.

2 Supporting Unanticipated Dynamic Software Evolution: Alternative Strategies based upon AO and Adaptation

The application of AOSD to adaptation is not a new idea [1, 21], and currently lots of works on adaptation are based on it. If we consider for instance [19], focused on the evolution of data models, we can observe that this work deals with the problems of structural and behavioural consistency arising after data model evolution. While structural consistency addresses the problem of accessing objects whose definition is no longer accessible after evolution, behavioural consistency refers to the problem of legacy applications having invalid references and method calls. The proposal of the authors is to encapsulate into aspects the adaptation code to access the evolved model, thus managing a more flexible result than those provided by approaches based on conventional class versioning.

Another proposal in this field is [9], which presents an architecture to manage the adaptation of non-functional concerns. The concerns that will be adaptable are given the shape of an aspect. The proposed architecture supports dynamic adaptation. In [18] it is shown how aspect oriented techniques can help adaptation in the context of pervasive computing environments. Again the idea is aspectizing those facets of the system which could be adapted. Similarly, [8] is focused on the Adaptive Object Model (AOM) architectural style, which supports adaptable systems not being adaptable itself. Using aspect oriented techniques the authors provide an adaptable AOM.

In [7], some suggestions to make join point models more open are proposed, in order to provide aspect oriented programming languages with a better support for adaptation. In [20], the Iguana/J architecture and programming model to support unanticipated dynamic adaptation is presented. Here, each functional class is associated with a set of adaptation classes which contain the adaptation code. The association is also specified in separated entities achieving improved flexibility.

Furthermore, aspect oriented techniques not only give support to code adaptation. In [14], it is shown how evolution and adaptability of software architectures can be managed combining aspect oriented techniques and coordination models. The different evolution needs are introduced as aspects for managing architectural adaptation.

But performing dynamic component adaptation requires information only available at runtime in most of the cases. If we want to take advantage of this information, we have to find a way to apply it at runtime as well in order to modify the behaviour of the components on the fly. We may consider two alternative strategies which have not been described in previous works:

Dynamic Aspect Generation: Adaptors are implemented by means of aspects which are generated, applied and removed at runtime as required. This approach increases the complexity of the infrastructure required for execution, demanding some non-trivial modifications to it, such as the inclusion and integration of new functionality (runtime aspect code generation and compilation). On the other hand, this approach would provide a high degree of flexibility in adaptor generation.

Dynamic Adaptor Management: Several precompiled aspects manage adaptation. In this approach, the different aspects form several managers which are able to retrieve and interpret the dynamic information required for adaptation. Different adaptors can be built using the algorithm described in [3], and managed specifically for each interaction between components as they join the context of the system.

So far, several efforts have been made in the community in order to develop platforms such as CAM/DAOP [16] or PROSE/MIDAS [17], which are already capable of performing dynamic aspect weaving, a mechanism that allows aspect code to be woven into an application at any point of its execution. This technique will enable the application of adapting aspects independently of the selected approach. Although the state of the art does not currently make Dynamic Aspect Generation a feasible approach, it is a promising choice to consider for future research. Dynamic Adaptor Management, on the other hand may be less flexible but suffices the requirements to perform dynamic adaptation, and the required infrastructure in comparison is much simpler. This justifies the adoption of this strategy for the first stage of this proposal. In the following sections, a new approach is described in which the main focus is put into using aspects for the implementation of the adaptation framework itself, rather than for aspectizing some facets of the system. Adaptation code is encapsulated into aspects, although not in a static way. On the contrary, aspects act as interpreters of the design information gathered from the components and as coordinators of the interaction between them.

3 Dynamic Adaptation Management Framework

3.1 System Architecture

When performing dynamic adaptation, signature and protocol information is required from the components being adapted to produce a consistent *mapping* or correspondence between their interfaces in order to solve potential mismatches. This can either be obtained from the components using techniques for the incorporation of metadata such as annotations [7], or semantic techniques [13] exploiting the already available information from the components, and inferring protocol related information such as order of message exchange in a similar fashion to OWL-S [15], used in the field of Web Services. While in the former a Grey-Box approach is taken for the extension of the system (increasing the accuracy of adaptation), the latter does not require the component to be specifically prepared. However, the available information may vary depending on the specific platform where adaptation is being performed, so a compromise may be necessary, such as taking a hybrid approach by adding some complimentary information to the components if it is required. Anyway, the construction of the aforementioned *mappings* falls out of the scope of this paper,

and it is an issue to discuss in itself in further work. For the purposes of this work, the *mapping* is considered to be already available, so the focus is put on the design of the aspect-based adaptation management framework. As it is depicted in Fig.2, the architecture of the system contains three basic functional modules in charge of the different tasks required for adaptation:

1. Interface Manager: Gathers information about the components' interfaces.
2. Adaptor Manager: Derives adaptors using the algorithm presented in [3] for the interaction between the components making use of the aforementioned mappings.
3. Coordination Manager: Coordinates the interaction between components, translating the messages based on the description of the adaptors previously derived.

The implementation of these tasks, grounded on the principles of AOP, exploit the join point model which enables clean message translation, since components do not need to be internally modified and pointcut definition provides a compact way to capture relevant events (component initialization and method invocation are of special interest). Although this framework relies on a standard join point definition language (a thing which usually implies suffering the consequences of structural and syntactical dependency from the base code [6, 11]), this does not affect the way in which the different managers operate, since the pointcut definitions used are trivial and do not include any specific syntactical nor structural patterns.

The implementation of these tasks as aspects, separating coordination from concerns such as adaptor generation, or interface description management grants a clear organization to the structure of the framework.

3.2 Interface Manager

Inspects the interfaces of the components as they join the context of the system, and keeps their description in an interface repository in order to use them later for mapping generation. For this purpose we will use reflection techniques. Upon initialization of the component c of class C, the manager checks for the existence of an entry for C in the repository, and if it does not exist, it creates one for it.

Since components usually exchange messages in a client-server manner, a complete description of both their offered and required interfaces is necessary. In the particular case of Java, the only information available is the description of the messages which belong to the offered interface M_o (through reflection), so the component must be complemented with a description of the signature of its required interface M_r. A complete description of both interfaces must include a minimum set of information for each method consisting on:

- Message (i.e. method) name.
- Ordered parameter names and types.

- Return value types.
- Exception types raised.

Component interfaces are also extended by including protocol information on their interface descriptions. The behaviour of the components can be specified by means of a Finite State Grammar (FSG) [22] which takes the set of available messages as input alphabet ($M_o \cup M_r$).

Summarizing, as it is observed in Fig.1, each of the entries in the interface repository contains a description of both offered and required interfaces and an automaton which specifies the protocol followed by the component.

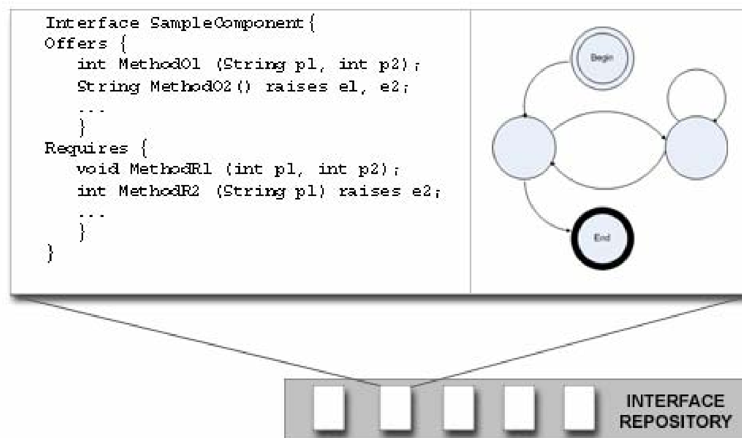

Fig. 1. Example of an interface description entry from the interface repository.

3.3 Adaptor Manager

It generates new adaptors as required by the conditions of the system. Once a component of class *S* joins the context, it may generate or receive one or several messages to/from other components. Every time one of these messages is generated or received, the manager captures it and checks if it is the first one consigned to or received from a target component class *T*. If that is the case, a mapping is produced between the source and target component classes, and subsequently an adaptor is automatically generated (Fig 3.c) making use of the algorithm described in [3]. This adaptor is stored in a repository and it will be used for interaction management between any pair of components of classes (*S*, *T*). This module will incorporate an inference engine based on pre-agreed ontologies explicitly defining resources, preconditions, and effects of processes, as well as domain related properties and relationships. In such a way, the system is provided with a machine-interpretable description of the semantics of the components. This enables the use of inference

techniques traditionally used in AI (knowledge representation, goal-oriented planning, logic, etc.) in order to infer relevant properties from the components and adapt them. Once generated, these adaptors allow syntactical adaptation providing message and parameter name translation, data conversion, and parameter reordering. They also provide a mechanism to perform protocol adaptation, storing messages whenever required for a delayed delivery, and establishing correspondences between them which can be one-to-one as well as one-to-many. By accessing the Adaptor Manager, engineers can supervise and tune the behaviour of the components by editing the *mappings* produced by the inference engine in order to fit specific needs. The characteristics of these *mappings* may also be constrained by manual introduction of contextual information in the engine. This capability enables a semi-automatic approach in which the engineer can easily evolve components worrying mostly about coarse-grained issues.

3.4 Coordination Manager

Monitors and translates all messages between components. Each time a component s_i sends a message to a component t_i, the manager translates it making use of the already available adaptor for (S, T) stored in the adaptor repository. A repository for session information is established in this manager in order to store specific information about the state of the components and their interaction. For each pair of interacting components (s_i, t_i), a session is created in the repository the first time s_i sends a message to t_i (see Fig 3.c). This session information is updated if necessary with each message between components. Session information will be publicly available to the mechanisms in the coordination manager since some interactions between components may influence that of others.

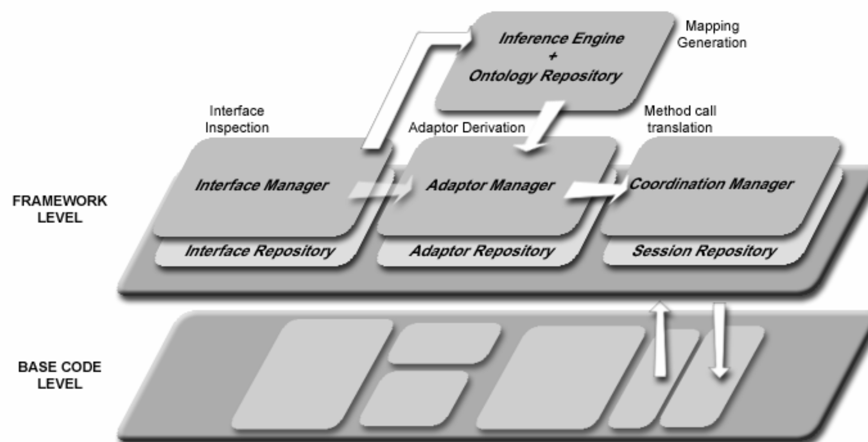

Fig. 2. Framework architecture diagram.

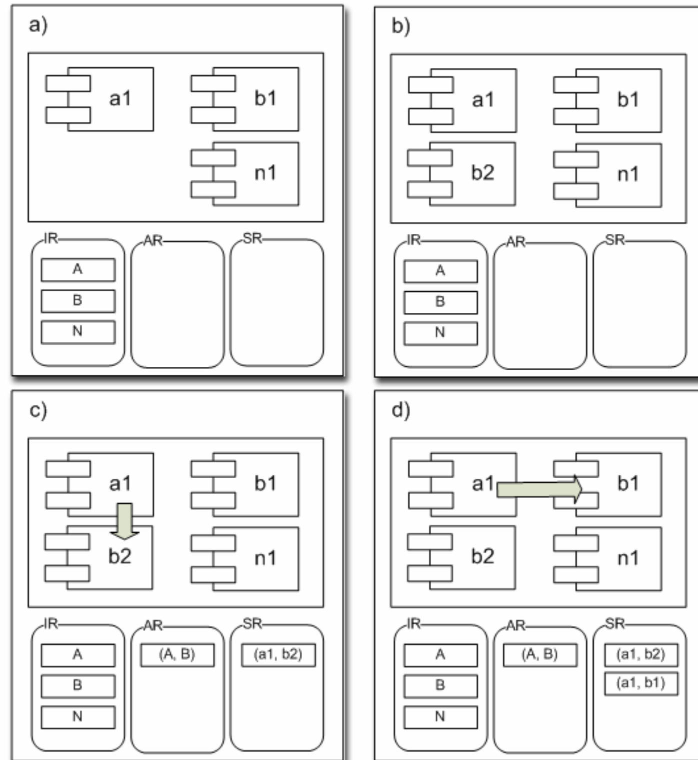

Fig. 3. Simple component interaction example: Components *a1,b1*, and *n1* join the context. Interfaces *A, B*, and *N* are stored in the interface repository **(a)**. Component *b2* joins the context **(b)**. *a1* sends a message to *b2*. Interfaces for *A* and *B* are mapped and adaptor (*A ,B*) is generated in the adaptor repository and a session entry for components (*a1,b2*) is created in the session pool. The message is then translated by the coordination manager **(c)**.*a1* sends a message to *b1*. A session entry is then created for components (*a1,b1*) in the session pool and the message translated by the coordination manager **(d)**.

3.5 Implementation Issues

In order to illustrate some of the issues related to the implementation of the proposal, AspectJ is used. This is a language level Java AOP extension which is highly representative of the AOP systems currently used. In this section some of the key structures and mechanisms provided to implement the functionality of the adaptation management framework are highlighted. Regarding the framework's design, a minimum set of pointcuts to define in order to provide the required functionality is:

Component initialization: It is satisfied whenever a new component enters the context of the system. It will be used by the interface manager in order to store interface related information.

Component invocation: Specifies all the messages sent from one component to another within the context of the system. Used by the adaptor manager for adaptor generation and by the coordination manager for session creation, message translation, and session information updating.

It is worth mentioning that since multiple aspects are present in the system, pieces of advice in the different aspects corresponding to each of the managers, may apply to a single join point. When this situation is given, the order in which advices are applied to the join point must be explicitly defined. This is the case of component invocation, which is used both by the adaptor and the coordination managers. In order to observe this order, AspectJ uses precedence rules to determine the sequence in which advices are applied. Aspects with higher precedence execute their before advice on a join point before the ones with lower precedence. When the method of a component is invoked, the sequence to follow is: **(a)** the adaptor manager checks if an adaptor needs to be generated. **(b)** The coordination manager checks if a session entry must be created, and **(c)** the coordination manager translates the message and updates session information. This translation is driven by the previously generated *mapping* and implemented through the join point model provided by AOP. This provides an elegant and non-invasive way of performing message translation.

AspectJ also provides mechanisms for source and target component identification through the use of `thisJoinPoint getThis()` and `getTarget()` methods. The coordination manager can monitor argument values in method invocations making use of the `getArguments()` method provided by `thisJoinPoint` as well. In order to obtain information related to methods such as exception, return, and parameter types, as well as argument and method names the `getSignature()` method provided by `thisJoinPointStaticPart` can be used.

Table 1. Pointcut definition and main API classes used for the framework.

Sample pointcut definition	
Component Initialization	`pointcut pcComponentInitialization() : staticinitialization(exp.adapt.component.*);`
Component Invocation	`pointcut pcComponentInvocation() : call(* exp.adapt.component.*.*(..));`
API structures and mechanisms	
Component Identification	`org.aspectj.lang.JoinPoint` `thisJoinPoint(getThis() and getTarget())`
Argument Values	`org.aspectj.lang.JoinPoint` `thisJoinPoint.getArguments();`
Method Information	`org.aspectj.lang.JoinPoint.StaticPart` `org.aspectj.lang.Signature` `(thisJoinPointStaticPart.getSignature())`
Class Identification and Interface Inspection	`java.lang.reflect.Class` `java.lang.reflect.Method`

Component class identification and interface inspection can be performed using the Java Reflection API. Through this API the class of each component can be obtained, along with information from it such as name, public attributes, and method signature description. It is worth noticing that parameter name information is not stored in standard Java `.class` files, so it is not retrievable using standard Java reflection. However, the AspectJ compiler does enrich compiled classes with that information. We will consider that we have that information readily available for our purposes.

4 Conclusions and open issues

In this paper, we have discussed the potential approaches to Aspect-Oriented Dynamic Component Adaptation in order to support Dynamic Component Evolution, as well as their advantages and drawbacks. We have justified the choice of dynamic adaptor management in a first approach and illustrated its foundational difference in comparison with previous proposals, which are usually focused on aspectizing some facets of the system. Then, a design for an adaptation management framework is proposed, showing its potential advantages as a tool to support the process of component evolution. In order to test this approach, a prototype in currently being developed in AspectJ. Although the platform does not support dynamic weaving, it is capable of performing load-time weaving, which is enough in order to prove the operational basis of the framework. The ontologies we are planning to use in this prototype will be stored in OWL. This will make it easier to create and read the ontologies since tools and libraries to process OWL are available. So far, only the signature and protocol levels have been tackled, and further study has to be performed related to mapping generation in order to provide suitable techniques for the semantic level as well. All this new functionality will be packed into the inference engine since all semantic level concerns are deeply interwoven with the process of mapping generation.

Dynamic component adaptation has proved to be a non-trivial problem which requires a vast amount of information about components for them to be successfully adapted in production environments. If it does not seem realistic on a first stage to take a White or even Crystal-Box approach considering the use of COTS products, adaptation, the most reasonable option seems to be taking a Grey-Box approach, by including key information on components (protocol, non functional concerns, etc.) in order to adapt them.

Although the current approach suffices the requirements to perform dynamic adaptation in simple cases, it is necessary to explore alternatives such as dynamic adaptor generation in further work in order to scale the problem to more complex scenarios.

References

1. Aksit M.,Tekinerdo Gan B., Bergmans L. Achieving Adaptability through Separation and Composition of Concerns. Muhlhauser M., Ed., Special Issues in Object-Oriented Programming, dpunkt, 1996, p. 12–23.
2. Allen R. and Garlan D. A formal basis for architectural connection. ACM Trans. on Software Engineering and Methodology, 6(3):213–49, 1997.
3. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. The Journal of Systems and Software. Special Issue on Automated Component-Based Software Engineering 74 (2005), pp. 45-54.
4. Canal, C., Fuentes, L., Pimentel, E., Troya, J.M., Vallecillo, A.: Adding roles to CORBA objects. IEEE Transactions on Software Engineering 29 (2003), pp. 242–260.
5. Canal, C., Murillo, J.M. and Poizat, P. Software Adaptation. in L'objet, 12(1):9-31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities. to appear. 2006.
6. Cazzola, W., J´ez´equel, J.M., Rashid, A. Semantic Join Point Models: Motivations, Notions and Requirements. In Proceedings of the Software Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT'06), Bonn, Germany, on 21st March 2006.
7. Cazzola, W., Pini, S. and Ancona, M. The Role of Design Information in Software Evolution. In Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05).
8. Dantas A., Borba P., Yoder J., Johnson R. Using Aspects to Make Adaptive Object-Models adaptable. Cazzola et al. in Reflection, AOP, and Meta-Data for Software Evolution, report num. Research report C-196, 2004, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology , p. 9–20.
9. David P.-C., Ledoux T. Towards a Framework for Self-Adaptative Component-Based Applications. Distributed Applications and Interoperable Systems (DAIS), vol. 2893 of Lecture Notes in Computer Science, Springer, 2003, p. 1–14.
10. Filman, Robert E., Friedman, Daniel P.: Aspect-Oriented Programming Is Quantification and Obliviousness. In Mehmet Aksit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors, Aspect-Oriented Software Development. Addison-Wesley, 2004.
11. Koppen, C., Störzer, M. PCDiff: Attacking the Fragile Pointcut Problem. In Proceedings of the European InteractiveWorkshop on Aspects in Software (EIWAS'04), Berlin, Germany, September 2004.
12. Magee J., Kramer J., and Giannakopoulou D. Behaviour analysis of software architectures. In Software Architecture, pages 35-49. Kluwer, 1999.
13. McIlraith, S.A., Martin, D.L.: Bringing semantics to Web Services. IEEE Intelligent Systems, 18(1):90-93, Jan/Feb, 2003.
14. Navasa A., Pérez M., Murillo J., "Aspect Modelling at Architecture Design", Morrison R., Oquendo F., Eds., European Workshop on Software Architecture (EWSA), vol. 3527 of Lecture Notes in Computer Science, Springer, 2005, p. 41–58.
15. "OWL-S: Semantic Markup for Web Services", The OWL Services Coalition (2004), http://www.daml.org/services.
16. Pinto, M.: CAM/DAOP: Component and Aspect Based Model and Platform, PhD thesis. Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga (2004).
17. Popovici, A., Frei, A., Alonso, G.: A proactive middleware platform for mobile computing. In: 4th ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil (2003)
18. Rashid A., Kortuem G. Adaptation as an Aspect in Pervasive Computing. Workshop on Building Software for Pervasive Computing at OOPSLA, 2004.

19. Rashid A., Sawyer, P., Pulvermueller, E. A Flexible Approach for Instance Adaptation during Class Versioning, Objects and Databases, vol. 1944 of Lecture Notes in Computer Science, Berlin, 2000, Springer, p. 101–113.
20. Redmond B., Cahill V. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. Object-Oriented Programming (ECOOP), vol. 2374 of Lecture Notes in Computer Science, Springer, 2002, p. 205–230.
21. Sánchez F., Hernández, J., Murillo, J. M., Pedraza E. Run-time adaptability of synchronization policies in concurrent object-oriented languages. Workshop on Aspect Oriented Programming at ECOOP (AOP), June 1998.
22. Yellin, D.M., Strom, R.E.: Protocol specification and component adaptors. ACM Transactions on Programming Languages and Systems 19(2) (1997)

An Aspect-Aware Outline Viewer (Work in progress)

Michihiro Horie Shigeru Chiba

Tokyo Institute of Technology
http://www.csg.is.titech.ac.jp

1 Introduction

An aspect-oriented programming (AOP) is for modularising a crosscutting concern so that it can be easily attached and detached to/from software. Because of this functionality, AOP is one of key technologies for enabling evolvable software. However, critics have been mentioning that AOP makes modular reasoning difficult since join points where an aspect and an object are connected to each other tend to spread over a whole program. Developers often have a problem finding join points specified by pointcut definitions in an aspect. To help developers, a tool such as AJDT has been developed.

For better modular reasoning in AOP, this paper presents a new interpretation of AOP, in which an aspect is an extension to an existing module but the extension may be effective only when the module is accessed from specific accessor modules. This interpretation should let developers consider an aspect is just an extension in the same sense that a subclass extends a super class and override some methods. Thus developers would be able to think that each module has an external interface and the internal implementation of the module is never directly accessed by other modules including an aspect.

To support AOP according to this interpretation, we have developed an Eclipse plugin. It is a programming tool for AspectJ and it shows an outline view of a class woven with an aspect. It presents how each method is extended by showing javadoc comments taken from the definitions of the class and the aspect. This tool gives developers a totally different illustration of AOP programs from AJDT, which is a standard programming tool for AspectJ. AJDT mainly shows the locations of join points (or join point shadows) selected by pointcuts. In other words, it only illustrates where an aspect and an object is connected to each other.

2 An event-based interpretation

A famous paper by Filman and Friedman [2] explained that AOP is quantification and obliviousness. According to their interpretation, program execution is modeled as a sequence of events, such as method calls and field accesses. An advice is an reaction to an event, *i.e.* a join point, selected by a pointcut. Thus,

to understand an AOP program, developers must know which events (*i.e.* join points) are selected for connecting an object and an aspect.

This event-based interpretation makes modular reasoning difficult since most of selected join points are part of the internal implementation of a module. For example, if a pointcut selects a join point representing a call to a setX method within a move method in a Line class (Figure 1), then that method call is part of the implementation of the move method and it should not be exposed to the outside of the Line class. Note that, here, the move method is not a *callee* method but a *caller* method. However, to understand the behavior of an aspect, developers must know the body of the move method contains the call to the setX method and it causes the execution of an advice body. The readers would think that the encapsulation principle is broken.

```
class Line {
  Point p1, p2;
  void move(int x, int y) {
       :
    p1.setX(newX);
       :
  }
    :
}
```

Fig. 1. The move method in Line calls the setX method in Point.

3 An extension-based interpretation

Although the encapsulation principle might seem broken in AOP, it is not really broken. To illustrate this fact, we present a different interpretation of AOP.

According to our interpretation, an aspect is an extension to a class although it might be effective only under some conditions. This is obviously acceptable if an aspect includes an advice associated with an execution pointcut, which selects the execution of a method body as a join point. Since the advice is executed together with that method body, the aspect can be regarded as an extension to the method body. Note that the extension does not break the encapsulation of the extended method body as an extension by inheritance does not. The extended method body is reused *as is* or the whole body is overridden.

An interesting case is an advice associated with a call pointcut, which selects the execution of a method-call expression at a caller side. Suppose that a move method in a Line class calls a setX method in a Point class and a call pointcut selects a call to setX (Figure 1). We explain that the advice associated with that call pointcut extends the behavior of the setX method in the Point class. An advice always extends the behavior of a *callee-side* method even if a pointcut

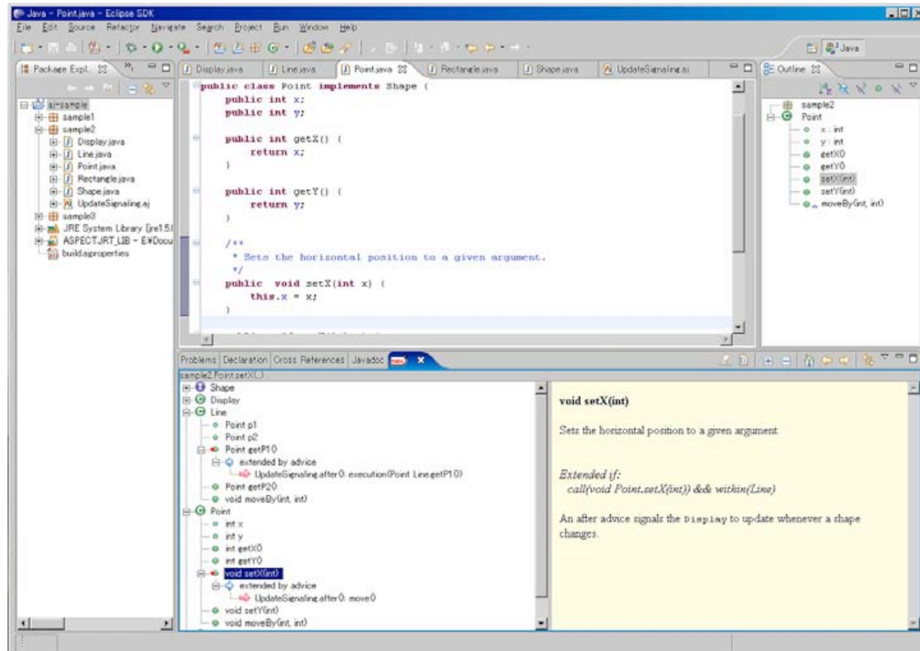

Fig. 2. Our outline viewer for a class extended by aspects (lower panel surrounded by a blue rectangle). It shows javadoc comments on both a method and an advice.

is call. It does not extend a caller-side method, for example, the move method in Line. Note that, under the event-based interpretation, that advice is often regarded as an extension to the caller-side method move.

A call pointcut can be combined with other pointcuts such as within and cflow. In this case, we explain that the behavior of a callee is extended by an aspect only when a caller satisfies the conditions specified by those other pointcuts such as within and cflow. For example, if a pointcut is the following:

```
call(void Point.setX(int)) && within(Line)
```

Then the advice associated with this pointcut extends the behavior of the setX method only when setX is called from a method declared in the Line class. This conditional extension cannot be implemented by subclassing; it needs AOP.

We similarly deal with get and set pointcuts as well. They extend the behavior of the fields that the pointcuts specify. For example, if a pointcut is get(int Point.xpos), then we consider that the advice associated with that get pointcut extends the behavior of the read access to the xpos field in Point. Without the extension, a read access to xpos simply returns the value of xpos. On the other hand, with the extension, a read access to xpos involves not only returning the value of xpos but also executing the associated advice.

void setX(int)
Sets the horizontal position to a given argument.

Extended if:
call(void Point.setX(int)) && within(Line)
An after advice signals the `Display` to update whenever a shape changes.

Fig. 3. The Javadoc comments on the setX method

4 Tool support

Our extension-based interpretation encourages developers to treat modules only through external interfaces even if aspects are woven with a program. The effects by aspects can be described as part of external interfaces. To support this idea, we have developed a AspectJ programming tool on top of the Eclipse IDE (Integrated Development Environment). It is an outline viewer of a class (Figure 2); it lists all the methods and fields declared in a specified class. If some of those methods and fields are extended by aspects, then our outline viewer also shows that fact. Furthermore, the outline viewer shows javadoc comments taken from both a class and an aspect. If developers select a method or a field extended by an aspect, then the outline viewer shows the javadoc comments on a pointcut and an advice as well as that method or field. For example, in Figure 2, the setX method in the Point class is selected. Thus, the outline viewer shows comments (a larger image is presented in Figure 3) in the right pane.

5 Concluding remarks

This paper presents the extension-based interpretation of AOP, in which an aspect is an extension to a *callee* class. Each advice in an aspect extends the behavior of a target method or a target field; it never extends a method at a caller (or accessor) side. If a pointcut includes a pointcut designator such as within and cflow, the extension is effective only when the execution context satisfies such a pointcut designator.

Our outline viewer presented in this paper helps programming with this interpretation. It is different from existing AspectJ tools such as AJDT, which supports the event-based interpretation. The outline view shown by our tool is similar to the aspect-aware interface [3]. Although our work shares basic ideas with the aspect-aware interface, we have further pursued appropriate concrete representation of modules in the presence of crosscutting concerns. For example, the article about the aspect-aware interface [3] does not mention how call, get, and set pointcuts should be reflected on a module interface. It does also not mention javadoc comments. Our outline viewer considers that an extension by an aspect is conditional if a pointcut includes within *etc.* This conditional exten-

sion is similar to the idea of Classbox/J [1] although Classbox/J is not an AOP language.

References

1. Bergel, A., S. Ducasse, and O. Nierstrasz, "Classbox/J: Controlling the Scope of Change in Java," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 2005.
2. Filman, R. E. and D. P. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," in *Aspect-Oriented Software Development* (R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, eds.), pp. 21–35, Addison-Wesley, 2005.
3. Kiczales, G. and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," in *Proc. of the Int'l Conf. on Software Engineering (ICSE'05)*, pp. 49–58, ACM Press, 2005.

Technological Limits for Software Evolution

Chairman: Hidehiko Masuhara, University of Tokyo

Solving Aspectual Semantic Conflicts in Resource-Aware Systems

Arturo Zambrano, Tomás Vera, and Silvia Gordillo

LIFIA, Facultad de Informática Universidad Nacional de La Plata
50 y 115 1er Piso
1900 La Plata, Argentina
[arturo, tomasv, gordillo]@lifia.info.unlp.edu.ar

Abstract. Aspects sometimes conflict between them in scenarios where they reify resource awareness concerns. These conflicts are the result of the scarcity of resources and the fact that, frequently, aspects are mutually oblivious. This kind of conflict can be solved by managing aspects according to the context. Obliviousness, even between aspects, can be retained and, at the same time, specific (per situation) conflict resolution strategies can be applied.

1 Introduction

Mobile applications must face a continuously changing environment. Resource and service availability can change dramatically during run-time. Then, resource-awareness is a must in such applications. Context and resource awareness are *per se* invasive, and prone to produce tangled designs and code. Because of this, a common practice is to isolate such behavior in the middleware layer. In [9] and [18] it has been suggested that the use of aspect orientation constitutes a means of decoupling context aware functionality from mobile applications.

The advanced separation of concerns provided by AOSD fits in the field of middleware in general [1,2,4], and it is specially suitable for implementing middleware for mobile context-aware systems. Aspects reifying different resource related concerns provide a modular way of handling them. On the other hand, aspects in resource-awareness middleware often compete for common (shared) resources they need. Conflicts are usual in this domain [16] and lead to aspect interactions which are related to the aspects' behaviour semantics (the way resources are utilised by aspects). These interactions cannot be detected just by syntactic means. As a result, they cannot be detected in compile time, since they depend on run-time conditions. Consequently, new approaches are needed to cope with this kind of conflicts.

In this work we will exemplify conflicts between aspects in a resource aware environment. We also present our approach for semantic-conflict resolution. This is based on the use of meta information attached to aspects, such metadata is used afterwards for conflict detection and aspect management.

This paper is organised as follows: Section 2 presents the motivation for our work. Section 3 summarises previous related research works. In Section 4 a

conflict resolution mechanism is proposed. Finally, we present our conclusions and future work in Section 6.

2 Aspectual Semantic Conflicts

2.1 Context

In an aspectual middleware for mobile applications, several aspects adapt application's behaviour to run-time conditions to ensure they use the available resources in the most effective way. This approach releases the application of resource management responsibilities, modularising behaviour that otherwise would be tangled with application's one.

We argue that even when no interference is a desirable state for aspects, this condition is not always held in runtime; since aspects for mobile client-side middleware implement a concern consuming (possibly) shared resources. Examples are provided in section 2.2.

It can be said that aspects should be represented in such a modular way that they do not affect other aspects or concerns. On the other hand, any behaviour added by aspects will consume resources; at least it will consume the processing cycles needed to execute its instructions. In some cases, this is not a problem, but in the context of mobile computing, where resources are scarce, a resource-conflict[1] could be a major problem. Therefore, it is a pragmatic problem that has to be considered when several aspects that manage resources are running in a mobile application.

The general idea is illustrated in figure 1 a). Each aspect manage a resource, the objective of the aspect is to perform the action indicated by the solid arrows, but its behavior produce a side-effect on another resource (dotted arrows). If the aspects and resources form a cycle, the result may be system with unstable behavior, as we will see later.

2.2 Examples

In this section we will exemplify some conflicting scenarios in the context of mobile computing. The examples show several aspects which interfere with each other while trying to accomplish their objectives.

Memory Saver Vs Battery Optimiser

Memory Saver Aspect monitors the memory usage by periodically checking the amount of free program memory. When it detects there is little available memory, this aspect forces all caches to flush their content.

Battery Optimiser Aspect is in charge of maximising battery lifespan. Since wireless network connections consume a lot of power, this aspect delays such

[1] In the context of this work, a conflict means the use of a given resource in an uncoordinated way, which may be dangerous for a system

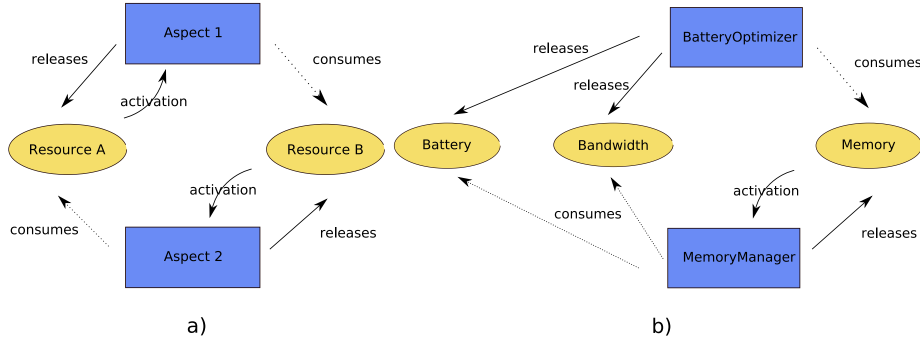

Fig. 1. a) Abstract conflict schema. b) A concrete conflict schema.

connections; that is, whenever the mobile client tries to send data to the server, the optimiser captures the outgoing data and stores it temporarily. When enough data has been collected, the optimiser performs a real network connection and sends all the stored data to the server.

As the reader can see, *Battery Optimiser* is affecting the resources it needs to perform optimisations, mainly memory, which is also the focus of the *Memory Saver* aspect.

Both *Memory Saver* and *Battery Optimiser* are oblivious to each other. Even though each aspect is aimed to work on its own concern, each one is influenced by the behaviour of the other.

It is important to note that the kind of conflict we are dealing with cannot be determined in compile or static weaving time. This example is illustrated in figure 1 b).

Network Optimiser Vs Security Vs Processing Time

Suppose that in the context of a wireless network, low traffic is desired in order to minimise packet loss problems and improve the response time. Also, as it is a wireless network, some security mechanism is needed, namely encryption. Conversely, encrypted messages are usually larger than their non-encrypted counterparts; therefore, they increase network traffic. Consequently we can have security but paying with a higher bandwith usage. In order to lighten this problem, we decide to compress messages before sending and decompress after receiving them. Now we have paid with CPU time.

A more sensible approach could be an adaptive one, where security is always provided using encryption, and compression is applied just when there is excessive network traffic and idle processor time.

Discussion From the previous paragraphs it is clear that conflicts among aspects exist, even if they are not working on the same joinpoints.

While considering aspect conflicts, it is important to keep in mind the notion of obliviousness [11]. Despite the fact that obliviousness is not a requirement for

an aspect oriented system, it is an important property that, if it is reached, it brings additional loose coupling. In this work we intentionally try to build context aware aspects that are mutually oblivious.

We argue that semantic conflicts can be solved by expressing the aspects' semantic without losing obliviousness. Aspects' semantics can be denoted through metadata; and modern programming languages, such as Java and C# provides means of expressing it. Therefore, it is fairly possible to develop metadata-based approaches, which can be easily implemented using the mentioned facilities.

3 Related Work

Recent research work in aspectual conflict detection has been developed by matching pointcuts syntactically[7]. Dounce et al. [10] propose a theoretical analysis framework to detect conflicts. However, little has been said regarding how to solve or avoid semantical aspectual conflicts; i.e. conflicts arising from the composition of behaviours that do not fit or are counterproductive, even when they might act on different joinpoints.

Concern Oriented Requirement Engineering techniques face the problem of conflicting concerns [14] at requirements level. In that work requirements are grouped into concerns, and the impact derived from the relationship between concerns is calculated. This impact is used to determine the existence of conflicts which can be solved by priorisation or renegotiation with the stakeholders. This approach may lead to a coarse grained aspect priorisation. In addition, this priorisation is fixed and applied to the whole system; since no specific situation customised priorisation is given. As it has been shown in Section 2.2, some conflicts may arise on runtime, and the time of their occurrence cannot be foreseen during requirements engineering phase.

Tessier et al. present, in [17], a model-based methodology which allows the detection of direct conflicts between aspects. In that work a taxonomy of conflicts is offered; the categorisation includes *Crosscutting Specifications*, *Aspect-Aspect Conflicts*, *Base-Aspect Conflicts* and *Concern-Concern Conflict*. Our work can be framed by the later category, in particular by the subcategory *Inconsistent Behaviour*, that refers to conflicts where one aspect can alter the state used by other aspects.

Bergmans [6] propose the use of annotations as a means of detecting conflicts among cross-cutting concerns. In his approach, conflicts can be detected when multiple concerns works on the same join point. As we previously said, our work aims to solve conflicts arising even when involved aspects work on different joinpoints.

This work is different with respect to previous conflict resolution approaches because it is focused on the semantic and behaviour of aspects, rather than their syntactic pointcut clashing.

4 Semantic Conflict Resolution for Resource-Awareness

4.1 Analysing Conflicting Situations

Despite the fact that conflicts cannot be completely foreseen using syntactic techniques, it is possible to anticipate conflicting situations by performing a domain analysis and reasoning about risky system situations.

In the context of *resource-awareness*, conflicting situations can be characterised as "malformed" combinations of resources' states. By "malformed", we mean situations where aspects executing normally, but in a non-coordinated way, can affect the proper system's behaviour. Following our first example, a conflicting situation arises when the system has little memory, and data cached by *Battery Optimiser* must be flushed very quickly. In this case, a sensible approach can be to deactivate the *Battery Optimiser* aspect. It is clear that if there was enough available memory, all aspects would run smoothly, so that no conflict would arise.

A list of conflicting situations must be constructed. Each situation must be described as [resource-state] pair list, which must be accompanied with corresponding corrective actions.

How to find those conflicting situations is outside the scope of this paper, and it is part of our related and future work.

4.2 Solving Conflicts

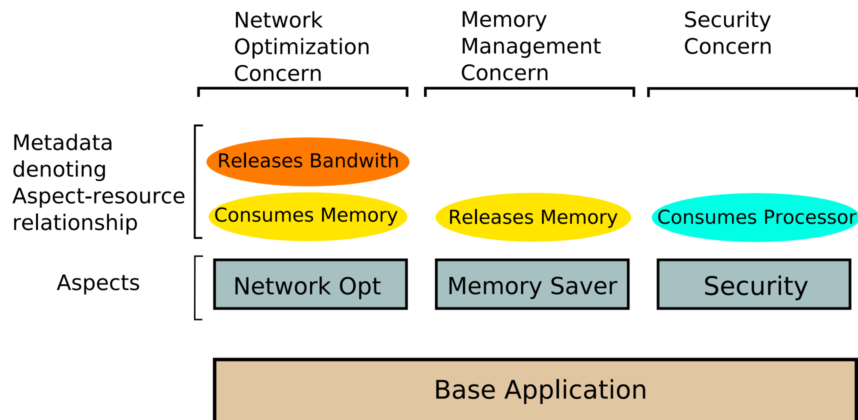

Fig. 2. Aspects' metadata indicating effects on resources.

In this section we present the corner-stones of our vision of conflict resolution.

Semantic Labels Java annotations [12] (and .Net attributes [15]) are powerful mechanisms that enable lightweight language extensions. They are used with multiple purposes such as attaching domain modelling information and enabling constraint enforcement. More recently, they have been used, along aspects, to demark and modularise crosscuting-cuttings concerns [13]. In order to explicit the use we make of annotations and to differentiate them from other general purpose annotations, we will call them *semantic labels* when they are utilised in the context of this work.

In order to solve/avoid runtime aspectual conflicts we propose the use of semantic labels. Semantic labels are descriptors added to aspects. These descriptors expose aspects' metadata indicating the kind of resources utilised by the aspect and the way they are affected (for instance, consumed or released), that is, the relationship between an aspect and the resources manipulated by it. In other words, semantic labels denote the role played by the aspect regarding a resource.

Figure 2 shows each aspect with their respective metadata, which describes the kind of operations the aspect performs on a resource.

Semantic labels provide a more abstract way of talking about aspects, as we shall see later. In fact, they define a discourse domain for aspects and resources.

Coordinator Aspect An extra aspect is necessary in order to control the execution of other aspects and solve conflicts. We call this aspect "Coordinator". Semantic labels are consumed by the *Coordinator* aspect in order to have a complete picture of aspects, resources and their relationship.

The *Coordinator* monitors resources' state looking for patterns indicating conflicting situations (Section 4.1), that is, certain state-resource pair patterns. Coordinator is supplied with a set of strategies. Each strategy is associated to a conflicting situation, and is defined as actions to be taken on the aspects involved. For example, a strategy can be defined as *switch off all memory consumers aspects*. Therefore, strategies are expressed as operations on aspects playing defined roles. Furthermore, since roles are used to express strategies, quantification can be achieved.

When a conflicting situation is found, the *Coordinator* applies the corresponding strategy. This means that it looks for the aspects playing the roles and performs some management operations on them. By performing the operations, the system behaviour is affected, and the conflicting situation eliminated.

Notice that, for sake of paper's length, we are talking about a coordinator aspect as a single module, when it is actually splitted in several parts.

4.3 Design Details

Figure 3 outlines the design of the proposed solution's prototype. The abstract ResourceHandler aspect provides the basic functionality that allows aspect management, that is switch-on/off behaviour.

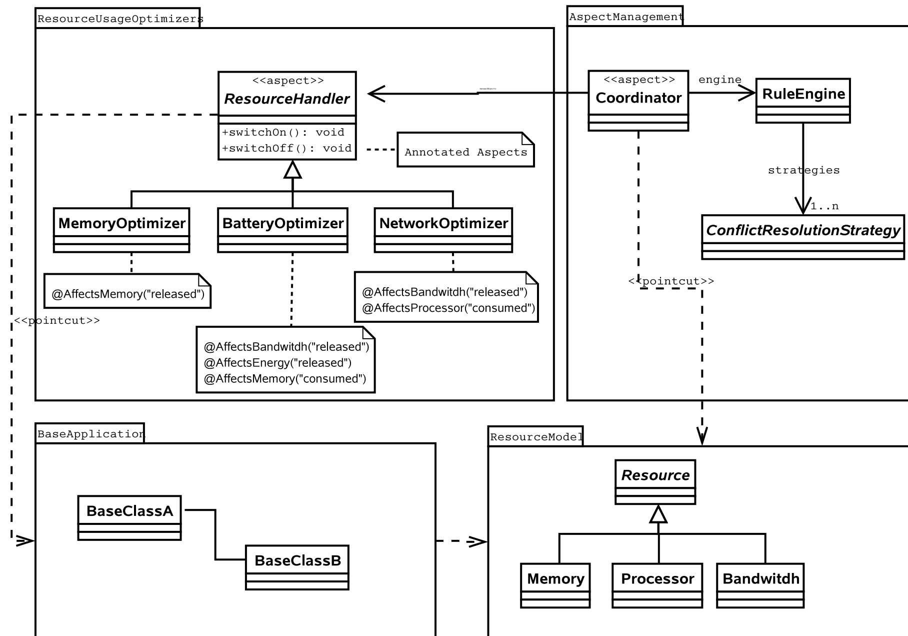

Fig. 3. Conflict Resolution Approach Class Diagram.

```
1    @AffectsMemory(CONSUMED)
2    @AffectsBattery(RELEASED)
3     public aspect BatteryOptimiser
```

Listado 1.1. Annotated Aspect

Subaspects are defined in order to perform several optimisations on the base program. These aspects must declare which resources they affect and the way it is done (denoted as UML comments [2] in the diagram). Such declarations are performed by annotating the aspects' code, as shown in Code Listing 1.1. Each annotation refers to a particular resource, and the parameter express how the resource is used.

The Coordinator aspect monitors the evolution of resources states. When a change is detected, a snapshot of resources states should be passed to the rule engine. Then, some strategies will be activated according to conflict-prone situations at hand. These active strategies perform corrective actions on the application's aspectual world. They can switch on/off aspects as necessary, so that conflict is neutralised. Aspects are not directly named by strategies, instead annotations are used to refer to aspects in an abstract form (see code listing 1.2).

[2] Since there is no unified way of expressing annotations in UML we follow one of the possibilities presented in [8]

```
1    rule "ControlMemory"
2     no-loop true
3    when
4     m : Resource(name == "memory",
5                  availability < 10 )
6    then
7     Coordinator.stop("@AffectsMemory
8                         (CONSUMED)")
```

Listado 1.2. A simple rule for aspect management

5 Prototype Implementation

In order to have a proof of concept we have implemented a prototype using Java Annotations [12], AspectJ [5] and JBoss Rules [3] as rule engine. In the implementation the coordinator intercepts the creation of each resource-management aspect. At this point it inspects the metadata and keep a reference for each aspect, so that it can be easily located when it is needed to perform a management operation.

The coordinator intercepts any resource state change. With this information it feeds the rule engine and fires the rules evaluation. When a rule matches its condition with the current resources' state combination it asks the coordinator to perform specific management operations on registered aspects.

6 Conclusions and Future Work

In this paper it has been shown how semantic conflicts can be solved. The foundations for semantic conflict resolution among aspects have been stated. They include the use of semantic labels for aspects, the early characterisation of conflicting situations, and the use of a coordinator aspect to detect and resolve runtime conflicting situations.

The proposed approach brings the benefits of coordinated aspectual behaviour. At the same time, obliviousness among aspects is preserved by using metadata.

Aspects can be designed and implemented separately and, later on, in an integration phase, their conflicts can be studied and strategies for solving them implemented. Since strategies are expressed in terms of aspects' metadata, strategies are loosely coupled to aspects. Therefore, they can be easily reused. Besides this, strategies define tailored priorisation for aspects in each specific situation, this feature contrasts with other fixed priorisation approaches such as [14].

Unlike other approaches, where a redesign of the aspects is required when a conflict is found [7,10], this approach allows the independent development of aspects, leaving conflict resolution isolated from the aspects.

Using this approach it is actually possible to prevent conflicting situations by building a set of strategies that carefully depict potential problematic situations and deactivate them before becoming a real problem.

These ideas have been illustrated in the field of resource aware, but they can be extrapolated to other situations where aspects manage a common base of resources. This approach has been develop to deal with non-core aspects. Working with core aspects require a careful and complete study of all possible situations, since manage such aspects may lead the system to stop complying its functionality. Our future work includes the generalisation of the presented approach in order to cope with aspectual conflicts in different usage contexts. Part of this work may involve the development of onthologies and the definition of conflict resolution strategies in terms of them.

References

1. Java aspect components. http://jac.objectweb.org/.
2. Jboss aop. http://www.jboss.org.
3. Jboss rules engine. http://www.jboss.com/products/rules.
4. Spring framework. http://www.springframework.org.
5. AspectJ project.
http://www.eclipse.org/aspectj/.
6. L. M. J. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. In J. Hannemann, R. Chitchyan, and A. Rashid, editors, *Analysis of Aspect-Oriented Software (ECOOP 2003)*, July 2003.
7. S. Casas, C. Marcos, V. Vanoli, H. Reinaga, L. Sierpe, J. Pryor, and C. Saldivia. Administración de conflictos entre aspectos en aspectj. In *Proceedings of the Fourth Argentine Symposium on Artificial Inteligence*, pages 1–11, 2005.
8. V. Cepa and S. Kloppenburg. Representing explicit attributes in uml. *7th International Workshop on Aspect-Oriented Modeling (AOM)*, 2005.
9. A. Dantas and P. Borba. Developing adaptive j2me applications using aspectj. *J. UCS*, 9(8):935–955, 2003.
10. R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 141–150. ACM Press, Mar. 2004.
11. R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. pages 21–35. Addison-Wesley, Boston, 2005.
12. JCP. A metadata facility for the javatm programming language, 2004. http://www.jcp.org/en/jsr/detail?id=175.
13. R. Laddad. Aop and metadata: A perfect match, March 2005. http://www-128.ibm.com/developerworks/java/library/j-aopwork3/.
14. A. M. D. Moreira, J. Araújo, and A. Rashid. A concern-oriented requirements engineering model. In O. Pastor and J. F. e Cunha, editors, *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.
15. MSDN. C # language specification - attribute specification. http://msdn.microsoft.com/library/default.asp?url =/library/en-us/csspec/html/vclrfcsharpspec_17_2.asp.

16. C. Shin and W. Wook. Conflict resolution method using context history for context-aware applications. In *First International Workshop on Exploiting Context History in Smart Environment. Pervasive 2005*, 2005.
17. F. Tessier, M. Badri, and L. Badri. A model-based detection of conflicts between crosscutting concerns: Towards a formal approach. In M. Huang, H. Mei, and J. Zhao, editors, *International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, Sept. 2004.
18. A. Zambrano, S. E. Gordillo, and I. Jaureguiberry. Aspect-based adaptation for ubiquitous software. In F. Crestani, M. D. Dunlop, and S. Mizzaro, editors, *Mobile HCI Workshop on Mobile and Ubiquitous Information Access*, volume 2954 of *Lecture Notes in Computer Science*, pages 215–226. Springer, 2003.

Statement Annotations for Fine-Grained Advising

Marc Eaddy Alfred Aho

Department of Computer Science
Columbia University
New York, NY 10027
{eaddy,aho}@cs.columbia.edu

Abstract. AspectJ-like languages are currently ineffective at modularizing *heterogeneous concerns* that are tightly coupled to the source code of the base program, such as logging, invariants, error handling, and optimization. This leads to complicated and fragile pointcuts and large numbers of highly-repetitive and incomprehensible aspects. We propose *statement annotations* as a robust mechanism for exposing the join points needed by heterogeneous concerns and for enabling declarative fine-grained advising.

We propose an extension to Java to support statement annotations and AspectJ's pointcut language to match them. This allows us to implement heterogeneous concerns using a combination of simple and robust aspects and explicit and local annotations. We illustrate this using a logging aspect that logs messages at specific locations in the source code. Statement annotations also simplify advising specific object instances, local variables, and statements. We demonstrate this using an aspect that traces method calls made to specific object instances and calls made from specific call sites.

Keywords: statement annotations, byte code annotations, fragile pointcut problem, logging problem, statement-level join points, instance-local advising

1 Introduction

Aspect-Oriented Programming (AOP) improves the separation of concerns by modularizing the code related to a concern that would otherwise be scattered throughout a program and tangled with the code related to other concerns. AspectJ-like languages are designed to modularize *homogeneous concerns*, which crosscut at module boundaries [17] and have a regular structure and common behavior [23].

1.1 Heterogeneous Concerns

Unfortunately, *heterogeneous concerns*, which exhibit irregular logic, are located at arbitrary places in the source code, and/or are highly coupled to the low-level structure of the code, are difficult to modularize using AspectJ-like languages [23]. Workarounds include creating complex and fragile pointcuts, writing a large number of highly-repetitive aspects, refactoring the base program to artificially expose the needed join points, or even abandoning AOP in favor of non-AOP solutions. The in-

ability for AOP languages to effectively express heterogeneous concerns severely limits the potential for AOP to separate crosscutting concerns.

This problem was first observed by researchers working on refactoring programs to use aspects. Murphy et al. needed to identify join points "in the middle of methods" to refactor graphical user interface code [15]. Since AspectJ could not capture these join points directly, their solution was to insert *dummy method calls* "which exist solely to provide access to the desired join points." This workaround has become so common in the AspectJ community that it is considered a de facto aspect refactoring idiom [8] [13].

In another refactoring exercise, Sullivan et al. observed that the logging concern was scattered across 180 different locations in the HyperCast application [20]. The arbitrary locations of the log messages required 20-30 complicated pointcut declarations that could only approximate the actual locations. Furthermore, the pointcuts were highly dependent on the structure of the underlying source code and would easily break when the code is modified. This is referred to as the *fragile pointcut problem* [14] [20]. Pointcut fragility in turn leads to the *AOSD-evolution paradox* [22], where programs written using AOP are actually harder to evolve, even though they are more modular. Other researchers have commented on the fragility of logging, optimization, and error handling aspects and the need for more robust and powerful pointcut languages [1] [5] [14] [17] [18] [19].

1.2 Our Approach: Statement Annotations

Statement annotations allow us to "name" any statement in a method body in a declarative fashion and attach arbitrary metadata. Statement annotations can be used to expose the join points needed to implement a heterogeneous concern or to perform fine-grained (instance- and statement-level) advising. This provides many benefits:

Pointcuts are simple and robust because they depend on semantically meaningful annotations instead of arbitrary program syntax or source locations [14] [3];

Advising is more fine-grained because advice can be applied to individual statements or object instances. While this is possible using other techniques, statement annotations provide a simple *declarative* way to perform fine-grained advising;

Advice is more reusable because it can access annotation metadata instead of hardcoding it;

Concerns can be easily integrated using a unified AOP solution instead of requiring a mixture of an AOP solution (for homogeneous concerns) and a non-AOP solution (for heterogeneous concerns);

The relationship between the base code and aspects is local and explicit, improving comprehensibility and maintainability [11].

Statement annotations must be invasively scattered throughout the base program and are thus crosscutting. We view this as an engineering tradeoff: we gain robustness, explicitness, and locality at the expense of obliviousness and modularity. Modularity is not sacrificed completely, however. Indeed, the parts of the concern that can be modularized *effectively* are specified in the aspect; the remainder is specified using annotations.

```
public class HelloWorld {
   public static void main(String args[]) {
      @State(State.Starting)
      System.out.println("Hello, World!");

   }
}
```

Listing 1. Statement annotation example.

1.3 Outline

In Section 2 we describe statement annotations, show how pointcut matching in AspectJ-like languages can be easily extended to match them, and how they can be used to implement the heterogeneous logging concern. In Section 3 we describe fine-grained advising and how it can be used to enable instance- and statement-level tracing. We discuss the tension between locality/explicitness and modularity with respect to statement annotations in Section 4 and related work in Section 5. Section 6 concludes.

2 Statement Annotations

Java and C#/.NET provide a flexible mechanism for allowing the programmer to attach annotations to program elements such as classes, methods, and fields. The annotations are stored in the class file as metadata and can be retrieved via reflection. However, these languages do not support annotating statements or byte code within the method body. This is unfortunate because statement annotations have the potential for a wide variety of uses:

Optimization – Statement annotations could potentially be used to enable OpenMP[1]-style parallel processing directives for Java. For example, some researchers have annotated *for-loops* to guide loop parallelization [2]. In addition, a compiler or other tool could annotate byte code with static analysis results to improve opportunities for optimization at JIT-time.

Bookkeeping – A form of byte code marking is used by AspectJ to prevent recursive weaving [10], by Steamloom to allow efficient enabling/disabling of advice at runtime [9], and by Spec#[2] to ignore injected code when performing static analysis. However, none of these systems allow arbitrary user-defined metadata to be associated with byte code.

---

[1] http://www.openmp.org
[2] http://research.microsoft.com/specsharp

Debugging and fault isolation – A debugger could selectively show or hide injected code based on the programmer's desired level of obliviousness. Injected byte code could be marked to indicate the source weaver or tool to improve fault isolation. Listing 1 shows an example of what a statement annotation could look like in Java.[3]

2.1 Statement Annotation Matching

In AspectJ 5, pointcut expressions may contain annotation patterns, but they can only match regular Java annotations defined on methods, fields, etc. We extend the pointcut expression matching algorithm to match statement annotations, effectively extending the AspectJ join point model to include any annotated statement. This gives us fine-grained control over when advice is applied and can simplify pointcut expressions that have many special cases.

Our proposed extension is very simple: Any pointcut expression that allows an annotation pattern should consider statement annotations in addition to regular annotations. For example, if method foo has the method annotation MA, and the statement annotation SA is used at a particular call-site, then the following AspectJ pointcuts will match that call-site:

```
call(@MA * *(..))
call(@SA * *(..))
@annotation(MA)
@annotation(SA)
```

This matching algorithm allows us to use the same annotation at either method- or statement-level granularity. If in the future we decide we need to be able to distinguish the two, we can introduce a new pointcut that only matches statement annotations.

2.2 Statement Annotations Simplify Complicated Pointcuts

In the HyperCast paper, Sullivan et al. observed that while it is sometimes possible to create pointcuts that match specific statements, they can become quite complicated [20]:

```
pointcut LogicalAddrChanged(Node node):
   set(LogicalAddress Node.MyLogicalAddress)
   && (withincode(void Node.messageArrived(Message))
      || withincode(void Node.timerExpired(Object))
      || withincode(void Node.resetNeighborhood()))
   && target(node);
```

Tourwé et al. observed that these kinds of complicated pointcuts arise because the pointcut language is too simplistic [22]. This hinders the evolvability of the base program because the pointcuts are likely to break if the underlying code changes.

[3] For the remainder of the paper we will use examples in AspectJ and Java. However, our extensions can be easily made to other languages.

This pointcut can be simplified if we are allowed to annotate the base program using method annotations:

```
pointcut LogicalAddrChanged(Node node):
   set(LogicalAddress Node.MyLogicalAddress)
   && withincode(@MyAnnotation * *(..))
   && target(node);
```

We can further simplify the pointcut by annotating the specific statements that require advising:[4]

```
pointcut LogicalAddrChanged(Node node):
   set(@MyAnnotation LogicalAddress
      Node.MyLogicalAddress)
   && target(node);
```

In general, annotations allow program elements to be explicitly named, eliminating the need for complicated pointcut expressions [3] or meta-programming. Annotations are explicit and collocated with the source code so they are more likely to be maintained as the program evolves. Pointcuts are only dependent on the annotations instead of the underlying structure of the code so they are more robust to changes and more reusable.

In the next section we show how to use a simple pointcut that matches statement annotations to implement the heterogeneous logging concern.

2.3 Heterogeneous Concern Example: Logging

Logging is the ability to record user-defined messages at specific points during the execution of the application. Logging is an example of a heterogeneous concern because its very nature is *ad hoc*—each log message is unique and often located at arbitrary points in the source code. Although both the tracing and logging concerns are complementary, and indeed often co-exist within the same application, logging is cumbersome to express using existing pointcut languages. This problem has become known colloquially as the *logging problem* [4] [12] [1] [21], however, it is a general problem that occurs anytime we try to capture a heterogeneous concern using AspectJ-like pointcut languages.

Listing 2 shows how the Note statement annotation can be used to expose specific join points within a method and attach arbitrary user-defined messages which can be logged by an aspect. When naming annotations, Kiczales and Mezini advice is to "choose a name that describes what is true about the points, rather than describing what a particular advice will do at those points." [11] Using their terminology, we

[4] Sullivan et al. took a different approach to simplify the pointcut that requires the base programmer to update a finite state machine (FSM) [20]. We believe our approach is complementary. Indeed, statement annotations can be used to update the FSM.

```
      ...several statements...
a.    @Note("Searching for plug-ins...")
      ...several statements...

      @Note("Entering long, but not infinite, loop")
b.    while (true) { ... }
      @Note("Loop exited successfully")
```

a-b. Using statement annotations to expose interesting events.

```
aspect LogNotesAspect {
   before(Note noteAnnotation) :
      @annotation(noteAnnotation) {

      System.out.println(noteAnnotation.value() +
         " [" + thisJoinPoint + "]");
   }
}
```

c. Aspect for logging notes.

Listing 2. Logging using statement annotations.

chose an "annotation-property"-like name for the annotation, e.g., *Note*, as opposed to an "annotation-call"-like name, e.g., *Log*.[5]

Statement Annotations versus Procedure Calls and Macros. The statement annotations in Listing 2 could be replaced with plain old procedure/method calls or macros. However, advice methods are more powerful because they have access to richer join point context information and more evolvable because they can access this context implicitly. Another difference is that plain old method calls are always called, and therefore always incur some overhead, even if the aspect is disabled. While macros can be used to alleviate this overhead by expanding to nothing at compile time, they do not help us at runtime.

Finally, resorting to method calls and macros to implement a crosscutting concern represents a fundamental failure of our AOP language. Some crosscutting concerns are implemented using AOP and some using non-AOP techniques even though the concerns may be very similar (e.g., tracing versus logging, error detection versus error handling) and may even share a common base implementation.

Statement Annotations versus Dummy Method Calls. In Listing 2a, an Event annotation appears at an arbitrary location in a method body. Without the annotation it would be difficult or impossible to identify the join point using an AspectJ pointcut, and, even if we could, the resulting pointcut would be very fragile [20]. An alternative to using a statement annotation is to insert a dummy method call at this location, a common AspectJ programming idiom.

[5] We thank Dean Wampler for suggesting this.

However, we find this unsatisfactory for several reasons. Dummy methods require a proliferation of empty methods which adversely affects design and code quality and can be confusing to the base programmer [15]. Statement annotations obviate the need for empty methods. Furthermore, statement annotations are less confusing because programmers familiar with annotation usage in Java and .NET are accustom to the interpretation of annotations at compile-time or postcompilation. Functionality that is woven into the base program as a result of the annotations will be less surprising, than, say, using around advice to replace empty method calls.

Unlike a statement annotation, a dummy method call is not directly associated with the next statement in the method body. Instead, it adds a new join point to the program. In the cases where the use of a statement annotation mirrors a method call or macro (aka an "annotation-call"), a dummy method call can be used instead. However, in the cases where a statement annotation is used to name and/or associate metadata with another statement (aka an "annotation-property"), dummy methods cannot be used. This makes dummy methods unsuitable for expressing some heterogeneous concerns such as local variable invariants and loop optimization hints [7] [6] and for performing fine-grained advising.

Statement Annotations Improve Statement-Level Pointcuts. Listing 2b shows statement annotations that bracket a *while-loop*. While AspectJ cannot match loops, some researchers have proposed extensions to the pointcut language to support matching statement-level join points of this kind [7] [6] [17]. While these proposed pointcuts make it easier to advise arbitrary statements, they are still fragile when they are used to identify specific statements.

Statement annotations allow individual statements to be discriminated without relying on the specific statement syntax. This means that if a *while-loop* is changed to a *for-loop* by the base programmer, the aspect will be unaffected. Furthermore, because statement annotations are explicit they are more likely to be kept up to date when the base source code is modified.

3 Declarative Fine-Grained Advising

Statement annotations are the first technique we are aware of for supporting declarative (as opposed to programmatic) instance- and statement-level advising. Examples of systems that support programmatic advising are Steamloom [9] and Eos [16]. The AspectJ *aspectOf* construct supports this to a lesser extent. The benefit of using statement annotations is that they only declare the programmer's intention. The actual advising is done by the aspect, where low-level decisions such as *if* and *how* instances will be advised can be deferred, rather than scattering these decisions throughout the base program, thus improving the separation of concerns.

```
class BankAccount {
   public void transferFundsTo(float amount,
            BankAccount destination) {
      // Trace all calls made to the ar object
      @Trace AuthorizationRequest ar =
        new AuthorizationRequest(this, destination);

      // Trace a call made at a specific call site
      @Trace destination.deposit(amount);
      …
```

a. Statement annotations for tracing instances and specific method calls.

```
aspect TraceAspect {
   static Set instances = new HashSet();

   after() returning(Object o) :
      call(@Trace *.new(..)) {

      instances.add(o);
   }
   before(Object o) : call(* *(..)) && target(o) &&
      if(instances.contains(o)) {

      System.out.println("Calling: " + thisJoinPoint);
   }

   before : call(@Trace * *(..)) {
      System.out.println("Calling: " + thisJoinPoint);
   }
}
```

b. Trace aspect using statement annotation matching.

Listing 3. Instance- and statement-level tracing. The statement annotations in **(a)** indicate that all calls to the `ar` instance should be traced as well as the `destination.deposit` method call The first advice in **(b)** adds instances created with the Trace annotation to the set. The second advice traces all calls to instances in the advised set. The third advice traces all calls originating from call-sites marked with the Trace annotation.

Tracing is the ability to record some or every method call[6] made during the execution of an application. Unlike logging, tracing is a homogeneous concern because it is highly structured in nature—the message format is consistent and the message is recorded at regular points in the execution of the application that are easily quantified using AspectJ-like pointcuts.

Annotations allow us to use tracing in a more heterogeneous fashion. For example, imagine we have a Trace annotation. We can use it as a method annotation and attach it to a particular method so that all calls to that method will be traced. Or we can use

[6] As well as other join points that can be easily expressed using AspectJ-like pointcut languages, such as field access.

it as a statement annotation to trace calls made to a specific object instance or to advise specific call-sites. This would be hard to do using existing pointcut languages.

Listing 3 shows an example of *instance-level advising*. The Trace statement annotation marks the `ar` object instance which will cause all method calls to it to be traced. The TraceAspect has advice that matches constructor calls that are marked with the Trace annotation and adds the object instance to the set of advised instances. Another advice matches any method call to an instance in the advised set.

In their paper on optimization aspects, Siadat et al. needed to advise a specific method call out of multiple calls to that method in the method body, but observed that AspectJ-like pointcut languages did not support this level of granularity [19]. Listing 3 shows an example of how this *statement-level advising* can be achieved using statement annotations.

Notice that the Tracing aspect is a normal AspectJ aspect. Without statement annotation matching support, the aspect will trace method calls to *any* instance of a class that has a Trace annotation on its constructor. With annotation support, we can narrow the focus to specific instances and call sites.

4 Discussion

Looking at the examples, it may appear that the logging and tracing concerns have not been modularized at all. However, the parts of these concerns that can be modularized *effectively*, including how messages are formatted, what join point context is used, and where messages are sent, are modularized by the aspect. For example, it would be relatively easy to change the aspects to send messages to a different location or to support asynchronous logging and tracing.

The parts of these concerns that cannot be modularized effectively, namely the user-defined messages, the locations in code, and which object instances and statements to advise, are better captured using statement annotations. The explicitness and locality of statement annotations makes it less likely that changes to the source code will invalidate the aspects, and makes it unnecessary for the programmer to have global system knowledge to assure that the pointcuts match correctly [14].

5 Related Work

Rho et al. present a fine-grained generic aspect language called LogicAJ2 [22]. Their pointcut language can match *arbitrary* declarations, statements, and expressions, and can bind to arbitrary join point context. The use of meta-variables in pointcuts, introductions and advice enables them to achieve heterogeneous, context-dependent effects. Unfortunately, identifying *specific* statements still requires referencing concrete base entities. This introduces dependencies which might break if base entities change.

By explicitly annotating the base code, using our statement annotations, for example, the base programmer could express these conceptual dependencies of aspects on

base entities in a less fragile way at the expense of giving up obliviousness and introducing scattering in the base code. This remains interesting future work.

6 Conclusion

We showed that by combining statement annotations and pointcuts, we can support instance- and statement-level advising, simplify pointcut expressions to make them more robust and reusable, and elegantly express heterogeneous concerns such as logging. We did this by proposing a natural extension to the AspectJ pointcut matching algorithm that is consistent with AspectJ's overall language philosophy.

For identifying individual statements and code locations, statement annotations are more robust than using complex pointcuts or meta-programming, and more elegant and obvious than using dummy methods. For advising specific object instances and statements, statement annotations are simpler and more succinct than using programmatic advising.

Unfortunately, annotations require intrusive (nonoblivious) changes to the base program and do not completely modularize concerns. It remains our future work to develop an AOP solution that can completely modularize heterogeneous concerns, possibly involving aspect visualization or automatic aspect refactoring, in a way that is easy to understand, maintain, and evolve.

Acknowledgements. We thank Boriana Ditcheva for creating a preprocessor for legalizing statement annotations for C#, Vibhav Garg for adding support for statement-level advising to Steamloom, and Kevin Sullivan and Yuanyuan Song for information about HyperCast. We also thank Matthew Arnold, Pascal Costanza, Günter Kniesel, Dean Wampler, and the RAM-SE workshop committee for their feedback.

References

1. R. Bodkin, A. Colyer, and J. Hugunin. "Applying AOP for Middleware Platform Independence," Practitioner Rpt.. In Proc. of Aspect-Oriented Soft. Dev. (AOSD'03), Mar 2003.
2. W. Cazzola, A. Cisternino, D. Colombo. "[a]C#: C# with a Customizable Code Annotation Mechanism," In Proc. of the Symposium on Applied Computing (SAC'05), Mar 2005.
3. V. Cepa and S. Kloppenburg. "Representing Explicit Attributes in UML," In Proc. of the Workshop on Aspect-Oriented Modeling (AOM'05), Mar 2005.
4. J. W. Cocula. "Can AOP really solve the 'logging problem'?," Online posting. AspectJ Disc. Forum. Apr 2003. http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg00383.html
5. A. Colyer, A. Clement, R. Bodkin, and J. Hugunin, "Using AspectJ for component integration in middleware," Practitioner Rpt.. In Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03), Oct 2003.
6. B. Harbulot and J. Gurd. "Using AspectJ to Separate Concerns in Parallel Scientific Java Code," In Proc. of Aspect-Oriented Soft. Dev. (AOSD'04), Mar 2004.
7. B. Harbulot and J. Gurd. "A Join Point for Loops in AspectJ," In Proc. of Aspect-Oriented Soft. Dev. (AOSD'06), Mar 2006.

8. W. Harrison, H. Ossher, and P. Tarr. “Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition,” IBM Rsch. Rpt. RC22685 (W0212-147), Dec 2002.
9. M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg and M. Krebs. “An Execution Layer for Aspect-Oriented Programming Languages,” In Proc. of Virtual Execution Environments (VEE'05), Jun 2005.
10. E. Hilsdale and J. Hugunin. “Advice weaving in AspectJ,” In Proc. of Aspect-Oriented Software Dev. (AOSD’04), Mar 2004.
11. G. Kiczales and M. Mezini. “Separation of Concerns with Procedures, Annotations, Advice and Pointcuts,” In Proc. of the European Conference on Object-Oriented Programming (ECOOP’05), Springer LNCS, Jul 2005.
12. G. Kiczales. “Can AOP really solve the ‘logging problem’?” Online posting. AspectJ Disc. Forum, Apr 2003. http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg00390.html
13. G. Kiczales. “General Best Practice Question,” Online posting. AspectJ Disc. Forum, Jul 2003. http://dev.eclipse.org/mhonarc/ lists/aspectj-users/msg00726.html
14. C. Koppen and M. Stoerzer. “PCDiff: Attacking the Fragile Pointcut Problem,” In Proc. of the European Interactive Workshop on Aspects in Software (EIWAS’04), Aug 2004.
15. G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. “Separating Features in Source Code: An Exploratory Study,” In Proc. of the International Conference on Software Engineering (ICSE’01), May 2001.
16. H. Rajan and K. Sullivan. “Eos: Instance-Level Aspects for Integrated System Design,” In Proc. of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE’03), Sept 2003.
17. H. Rajan and K. Sullivan. “Generalizing AOP for Aspect-Oriented Testing,” In Proc. of Aspect-Oriented Soft. Dev. (AOSD’05), Mar 2005.
18. T. Rho, G. Kniesel, and M. Appeltauer. “Fine-Grained Generic Aspects,” in Proc. of the AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL’06), March 2006.
19. J. Siadat, R. J. Walker, and C. Kiddle. “Optimization Aspects in Network Simulation,” In Proc. of Aspect-Oriented Soft. Dev. (AOSD’06), Mar 2006.
20. K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, “Information Hiding Interfaces for Aspect-Oriented Design,” In Proc. of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE’05), Sept 2005.
21. K. Sullivan. “Handling Logging and Tracing Concerns,” Online posting. AOSD.NET Disc. Forum, Jul 2005. http://aosd.net/pipermail/discuss_aosd.net/2005-July/001621.html
22. T. Tourwé, J. Brichau, and K. Gybels. “On the Existence of the AOSD-Evolution Paradox,” In Proc. of the AOSD 2003 Workshop on Software Engineering Properties of Languages for Aspect Technologies, Mar 2003.
23. M. Trifu and V. Kuttruff, “Capturing Nontrivial Concerns in Object-Oriented Software,” In Proc. of the Working Conference on Reverse Engineering (WCRE'05), Nov 2005.

Dynamic Refactorings: improving the program structure at runtime

Peter Ebraert ⋆ and Theo D'Hondt

Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium

Abstract. Many software systems must always stay operational, and cannot be shutdown in order to adapt them to new requirements. For such systems, dynamic software evolution techniques are needed. In this paper, we show how a restructuring technique – called refactoring – can be applied on running systems in order to facilitate future evolutions. We extend the pre- and post-conditions of the basic refactorings in order to ensure application consistency before and after the restructuring takes place.

1 Introduction

People always say that you should never change a system that is working fine. However, even if a software system seems to work properly from a user's point of view, it may be difficult to maintain or adapt from a developer's point of view. As such, it may be very cumbersome to evolve the system by adding a new feature, fixing a bug or porting the system to a new environment [1].

In all these situations where a software system is not flexible enough to allow a certain change, the technique of *software refactoring* can be used [2]. According to Fowler [3], a refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour". Refactorings can be used to simplify the structure of a software system in order to prepare it for a certain evolution step.

Now suppose we have a running system, and we would like to evolve it without shutting it down. This is a much bigger challenge since there are considerably more constraints on the running system [4]. Refactoring techniques would be very useful here too. For example, by reducing the coupling between objects in a running system, we could at the same time increase the system performance (from a user point of view) and its understandibility and evolvability (from a developer point of view).

Until now, refactorings have only been investigated in the context of source code restructuring [5]. The main contribution of this paper is to show the use and feasibility of applying *dynamic refactorings*, i.e., refactorings that modify a running system.

⋆ Author funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)"

The remainder of the paper is structured as follows. In the next section, we introduce the notion of atomic change sequences, which we will use throughout this paper for expressing refactorings. Every atomic change from such a sequence has a set of pre- and post-conditions. The pre-conditions represent some clauses that must hold for allowing the change. The post-conditions are clauses that hold after the change has been carried out. Section 3 presents an approach in which the pre- and post-conditions of the ordinary refactorings are extended in order to meet with the extra requirements that come with the dynamicity. In section 4 we provide a glimpse on how this approach can be applied in Visual Works, an IDE for the Smalltalk programming language. We conclude by evaluating our approach on a conceptual level in section 5.

2 Atomic Change Sequence

Refactoring object-oriented programs typically replaces a set of classes C_1, C_2, C_3, ... by their new versions C'_1, C'_2, C'_3,... We use the notation ΔC_i to denote the difference between C_i and C'_i. In this section, we first define a set of atomic changes that can be used as the building blocks for specifying the *atomic change sequence* – the ΔC that must be applied in an atomic way in order to apply the corresponding refactoring.

Most of the differences we want to express can be represented by methods and instance variables. This is why our meta-object protocol currently implements the set of atomic change transactions that is shown in table 1. In the future we intend to extend this set to cover a more realistic set of applications.

Scope	Atomic Change	Explanation
Class	AC	Add an empty Class.
	DC	Delete an empty Class.
	CC	Changes a Class name.
Variable	AV	Adds an instance variable to a class.
	DV	Remove an instance variable from a class.
Method	AM	Adds a method to a class.
	DM	Deletes a method from a class.
	CM	Changes the implementation of a method.
	ML	Change the Method Lookup.

Table 1. Atomic Changes

The most simple atomic changes incorporate the ones on the scope of classes: adding empty classes (AC), deleting empty classes (DC), and changing a class name (CC) have a small impact on the system and can be performed without too many constraints. The only pre-requisit of the AC and CC is that name-clashes should be avoided, and only be tolerated only when intended (in case of polymorphism).

The changes on the scope of instance variables are a bit harder. As a result of an added variable (AV), all objects that are instance of this class have a new

variable they can use to store values. By default, the value will be set to `nil`. A deleted variable (DV), deletes the variable in all the instances of the class. This is a dangerous operation as it could lead to runtime exceptions, if the variable is still used somewhere. That is why this atomic change has a prerequisite which states that the variable is not used throughout the system. Note that there is no operation of modifying a variable, as that can easily be modeled by deleting and adding the variable.

Finally, there are the changes that affect the methods. As a result of an added method (AM), all objects that are instance of this class will automatically understand this new method thanks to the languages method lookup mechanism. When a delete method (DM) is applied, all instances of this class may no longer understand this method. Hence, one should be very careful with this operation as it can give rise to runtime exceptions. The same counts for a changed method (CM), as this also has an impact on all objects that are instance of this class or one of its subclasses. This is why a CM and a DM have the prerequisites that the method is either still visible (somewhere up the inheritance chain), or either not called anywhere in the system.

3 Proposed solution

The following section describes the extension we want to make to the already known refactorings so that they can be safely applied to running systems. We start by explaining two basic refactorings and show how they are characterized in [3]. We show that the mechanics of applying the refactorings – as they are presented by Fowler – are not sufficient for applying the refactorings in a safe way on a running system. We then introduce extra prerequisites which ensure safety, and exemplify them by means of the corresponding two dynamic refactorings. We conclude the section by showing how the dynamic refactorings can be carried out on a running system.

3.1 Static Refactorings

In [3], Fowler presents a catalog of frequent refactorings. Every refactoring consists of a name, a motivation and the mechanics for applying the refactoring. We chose to explain our approach on 2 refactorings: "Pull Up Method" ([3] page 322) and "Push Down Method" ([3] page 328).

Pull Up Method Fowler introduced this refactoring for getting rid of unneeded code duplication. Its idea is to lift the common behavior of some classes to a common superclass as shown in figure 1. In table 2 we show the mechanics of the refactoring (represented as an atomic change sequence). As the pulled up method remains visible for all the subclasses and may even become visible for more classes, there can only be an addition of behavior. Every step has its pre- and post-conditions which must hold, for the refactoring to be valid. We can see that an AM leads to the *visibility* of that method in the class and its subclasses.

We also see that before a DM, we must assure that the method is either not called anymore, or either visible in one of the superclasses. Those requirements are captured in the pre- and post-conditions of the atomic changes.

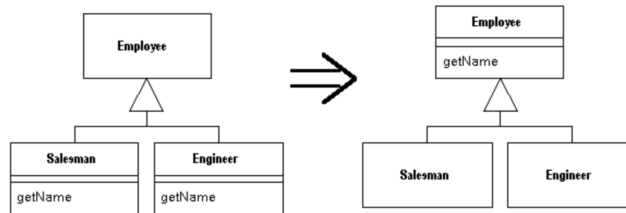

Fig. 1. Pull Up Method.

Place	Pre	Change	Parameters	Post
Employee		AM	"getName"	visible
Salesmen	no callers or visible	DM	"getName"	
Engineer	no callers or visible	DM	"getName"	

Table 2. The atomic change set for the Pull Up Method

Push Down Method Fowler introduced this refactoring for making the system behave in a more logical way, by specifying the behavior in the place where it makes more sense. Figure 2 shows that the refactoring is used for moving some behavior from a super class to a subclass. In table 3 we show the mechanics (represented as an atomic change sequence). Pushing down a method can be seen as a removal of behavior for all the classes in which the method is not visible anymore.

Place	Pre	Change	Parameters	Post
Salesmen		AM	"getName"	visible
Engineer		AM	"getName"	visible
Employee	no callers or visible	DM	"getName"	
Engineer	no callers or visible	DM	"getName"	

Table 3. The atomic change set for the Push Down Method

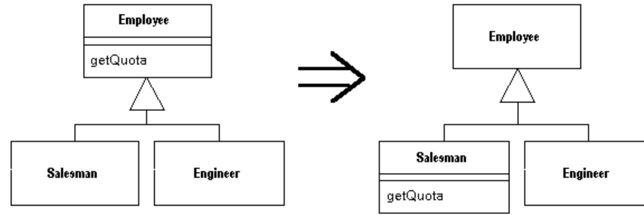

Fig. 2. Push Down Method.

3.2 Extra needs for dynamicity

The difference between stopped and running systems lies in the system state, which is only incorporated in the latter. In fact, a running system can be seen as moving from one consistent state to another while the processing of transactions goes on. A consistent state is a state from which the application can continue processing normally, without processing to an error state. When applying refactorings at runtime, we should always make sure that the application state remains consistent *before* and *after* the application of the update.

For ensuring state consistency before the application, we must make sure that the affected classes are in a *quiescent* status. An object in a quiescent status was proven to remain in a consistent state [6]. An object is in a quiescent status if: (i) it is not currently engaged in a transaction that it initiated, (ii) it will not initiate new transactions, (iii) it is not currently engaged in servicing a transaction, and (iv) no transactions have been or will be initiated by other objects which require service from this objects [6]. Theoretically, quiescence is achieved by adding extra preconditions which must hold before a refactoring can be applied. Practically, those preconditions are met by deactivating all objects that are affected by the refactoring. The deactivation and activation itself are added to the atomic change sequence of the refactoring. The post-condition of a deactivation is that the object is in a quiescent status $Q(O)$. The post-condition of an activation is that the object is in an active status $A(O)$.

Ensuring state consistency after the update clearly depends on the update itself. In our case, the updates only consist of refactorings. Since Fowler defined a refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour", we are by definition only making structural changes. If we make those changes in a correct way, state consistency can be easily assured.

3.3 Dynamic Refactorings

Extending the atomic change sequence of a refactoring is a process that can be automated. We need to add pre-conditions, postconditions, and actions that make sure that those pre- and post-conditions are eventually met. In general,

we first establish the set of all classes that are affected by the refactoring, and for each of them, we add a quiescence pre-condition. In practice, however, this process can be optimized, since quiescence is not needed for certain classes that are involved in the refactoring. For example adding a method to a class can never lead to run-time errors. We now exemplify the extension of the atomic change sequences by showing the two refactorings that were presented before.

Dynamic Pull Up Method In table 4 we show the mechanics of the dynamic refactoring (represented as an atomic change sequence). We can see that the actual refactoring is performed when the affected classes reside in a quiescent status and that it is surrounded by actions that ensure quiescence.

Place	Pre	Atomic cha.	Parameters	Post
Salesmen	A(Salesmen)	Deactivate		Q(Salesmen)
Employee	A(Employee)	Deactivate		Q(Employee)
Engineer	A(Engineer)	Deactivate		Q(Engineer)
Employee		AM	"getName"	visible
Salesmen	Q(Salesmen), no callers or visible	DM	"getName"	
Engineer	Q(Engineer), no callers or visible	DM	"getName"	
Salesmen	Q(Salesmen)	Activate		A(Salesmen)
Employee	Q(Employee)	Activate		A(Employee)
Engineer	Q(Engineer)	Activate		A(Engineer)

Table 4. The atomic change set for the Dynamic Pull Up Method

Dynamic Push Down Method In table 5 we show the mechanics of the dynamic refactoring (represented as an atomic change sequence). Again, we can see that the actual refactoring is performed when the affected classes reside in a quiescent status and that it is surrounded by actions that ensure quiescence.

3.4 Carrying out the refactoring

From the moment we have the change transaction sequence that corresponds to a certain refactoring, we can start carrying out these changes on the running system. The changes are applied one by one on the running system, but in an atomic way (or all changes are applied, or none of them is applied). While most of the changes can be done transparently, some may require the programmer's interference. This is the case when there is a state involved, that needs to be preserved. Concretely, when an instance variable is deleted or modified, the question arises what has to happen with the value of that instance variable. Either the value can be ignored, or its is needed later in a new instance variable that will be added. Consequently, when an instance variable is added, the programmer is also requested to interfere, and to tell wether the variable should

Place	Pre	Atomic cha.	Parameters	Post
Salesmen	A(Salesmen)	Deactivate		Q(Salesmen)
Employee	A(Employee)	Deactivate		Q(Employee)
Engineer	A(Engineer)	Deactivate		Q(Engineer)
Salesmen		AM	"getName"	visible
Engineer		AM	"getName"	visible
Employee	Q(Employee), no callers or visible	DM	"getName"	
Engineer	Q(Engineer), no callers or visible	DM	"getName"	
Salesmen	Q(Salesmen)	Activate		A(Salesmen)
Employee	Q(Employee)	Activate		A(Employee)
Engineer	Q(Engineer)	Activate		A(Engineer)

Table 5. The atomic change set for the Dynamic Push Down Method

be initialized with a certain value. For example, using Euros instead of Belgian Francs in our bank accounts requires us to use the following formula: 'take the old value and multiply it by 40,3399, and use it as the new value'. For methods in class-based systems, things are much simpler. Because methods are only referenced through the class itself, adapting them on the class level does the job. Making these changes is done in practice by using interceptive techniques [7], which incorporate all the atomic changes that are specified in the meta-object protocol.

Ensuring quiescence is the hardest part, and consists of two phases. First, we need to find all the affected objects. In a class-based language, the set of affected objects of a change on class C consists of the class itself, its subclasses and all instances of those classes. Practically, this set can be assembled by using introspective techniques [7], which allow us to query a class for all its subclasses and instances.

In the second phase we have to ensure for each of the affected objects, that: (i) it is not currently engaged in a transaction that it initiated, (ii) it will not initiate new transactions, (iii) it is not currently engaged in servicing a transaction, and (iv) no transactions have been or will be initiated by other objects which require service from this objects. In class-based programming languages, (i) and (iii) can be assured by making sure that the object is not on the runtime stack. (ii) and (iv) can be assured by making sure that no messages will be send to the affected object.

4 Experimental Setup

Because the focus of this paper is on refactorings, we restrict ourselves to class-based object-oriented languages. Object-oriented languages like Java are excluded because of the limitations of their reflective capabilities. Smalltalk, on the other hand, is fully reflective: everything is an object, and can thus be taken apart, queried for information and possibly be modified. Even messages are objects, and can thus be monitored and modified when they are sent or received

[8]. This is why we chose Smalltalk as the language in which we plan to conduct the experiments.

The Smalltalk development environment is very dynamic, in the sense that developing smalltalk code happens in an incremental way, by inspecting the newly created classes and objects. This is why developing Smalltalk programs actually happens at runtime. The Smalltalk Refactoring Browser provides support for static refactorings [9]. But because it does not check for the extra requirements that we have presented in this paper, it sometimes fails to ensure consistency. That is why we plan to extend the Smalltalk Refactoring Browser with the support for dynamic refactorings. We will make sure that, before a refactoring is performed, the refactoring browser will check for the additional requirements, making sure that the system remains in a consistent state.

5 Evaluation

The main benefit of applying refactorings at runtime is the preservation of the state and object identity, as we will keep on working on the same (already existing) class C. Replacing a class C would involve the creation of C', the swapping of all relations from C to C', the deletion of C and the mapping of the state from C to C'. Evolving the existing C component to C' only involves the creation of C' and the propagation of the changes on C. This implies that there will be no more relation swapping problems and less state mapping problems.

An auxiliary benefit of applying the refactorings at runtime is the presence of *runtime information*. Using this information, we do not only depend on *static analysis* for finding wether certain pre-conditions hold. In fact, the dynamic information can help us to overcome the two major shortcoming of the static analysis. First, because the calculating of some of the pre-conditions can be reduced to the *halting problem* [10], static techniques must make approximations for finding wether those pre-conditions hold. Dynamic information can help to improve the approximations. Second, doing static analysis in a dynamically typed language is always hard, because of the *lack of type information*. A solution to the problem lies in making a liberal approximation on what the type of some variable is. This is done by executing some scenario's and by collecting the dynamic information as the program is executed. Since it is impossible to observe every possible run of the program, the result is still an approximation depending on all the scenarios that were executed.

Most of this research is based on the Refactoring book of Fowler [3] in which he presents for each refactoring, a set of pre-conditions that would ensure that the refactoring would preserve behavior. The refactorings are defined in terms of C++, but many of them are applicable to other OO languages. The only refactorings that are not applicable to Smalltalk are the ones dealing with types and access privileges, since Smalltalk does not have these features. This explains why we will not be able to implement those refactorings in practice.

While it is clear that this approach is impossible without reflection, reflection itself also hinders the approach. Ensuring consistency on a program that uses

reflection is a lot harder then on a program that does not allow reflection. So in a sense reflection can be seen as both our best friend, and our enemy. In this position paper, we presented two static refactorings and their dynamic equivalents. It is our goal to present a dynamic equivalent for all of the refactorings that were identified in. For doing so, we will have to extend our metaobject protocol and maybe introduce new prerequisites. This, however, is subject for further investigation.

6 Conclusion

In some cases, software systems can not be turned off for carrying out an evolution. This triggers the need for supporting dynamic evolution. We suggested to apply dynamic refactorings to improve the runtime component structure of object-oriented software systems for easing future evolution. The approach relies on the reflective properties of the underlying programming language in order to modify the application's behavior.

We start from the static refactorings that were presented by Fowler in [3], and extend their mechanics in such a way that they can be applied on a running system without braking consistency. For doing that, we use the quiescence property, that was introduced in [6] as a sufficient condition for consistency. We add quiescence as a prerequisite for the refactorings and show how this can be implemented in a class-based environment that has full reflective capabilities. We envision the extension of the Smalltalk Refactoring Browser for experimenting with the dynamic refactorings.

References

1. Lientz, B., Swanson, E.: Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. Addison-Wesley (1980)
2. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Phd thesis, University of Illinois at Urbana Champaign (1992)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
4. Ebraert, P., Vandewoude, Y., D'Hondt, T., Berbers, Y.: Pitfalls in unanticipated dynamic software evolution. In Cazzola, W., Chiba, S., Saake, G., Tourwé, T., eds.: RAM-SE'05 – ECOOP'05 Workshop on Reflection, AOP and Meta-Data for Software Evolution, University of Magdeburg, Germany, Preprint No. 9 (2005)
5. Mens, T., Tourw'e, T.: A survey of software refactoring. IEEE Transactions on Software Engineering **30** (2004) 126–139
6. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering **16** (1990) 1293–1306
7. Maes, P.: Computational Reflection. PhD thesis, Artificial Intelligence Laboratory, Vrije Universiteit Brussel (1987)
8. Messick, S.L., Beck, K.L.: Active variables in smalltalk-80. In: Technical Report CR-85-09, Computer Research Lab, Tektronix (1985)

9. Roberts, D., Brant, J., Johnson, R.: A refactoring tool for smalltalk. Theory and Practice of Object Systems **3** (1997) 253–263
10. Hopcroft, J.E., Motwani, R., Rotwani, Ullman, J.D.: Introduction to Automata Theory, Languages and Computability. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)

Implementing Bounded Aspect Quantification in AspectJ

Christian Kästner, Sven Apel, and Gunter Saake

Department of Computer Science
University of Magdeburg, Germany
email: {kaestner,apel,saake}@iti.cs.uni-magdeburg.de

Abstract. The integration of aspects into the methodology of *stepwise software development (SWD)* is still an open issue. This paper focuses on the global quantification mechanism of nowadays aspect-oriented languages that contradicts basic principles of SWD. 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 [10]. 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 [7, 19], database systems [17], and operating systems [12], several issues remain controversial or simply not addressed.

This paper aims at the connection of AOP and the methodology of *stepwise software development (SWD)* [18, 16]. The idea behind SWD is to evolve a program starting from a minimal base by successively applying refinements that encapsulate 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 [13, 14].

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

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]; 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 [13, 14], we address several issues that arise from the practical implementation, i.e. in AspectJ. We discuss several alternatives to realize such a 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 [18, 16]. Several studies have shown that this circumstance is directly responsible for inadvertent aspect interactions and an unpredictable behavior in evolving software [14, 15, 9, 8]

In order to address this issue, Lopez-Herrejon et al. proposed an approach to aspect composition [14]. 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 [14, 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).

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 7-10). 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, it affects the color feature

[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/arj/
[3] http://abc.comlab.ox.ac.uk/

```
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 }
```

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

```
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) : execution(* Point*.set*(..)) && target(p);
8    after(Point p) : setCoordinates(p) {
9      p.cntr++;
10   }
11 }
```

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

applied subsequently. It advises the *setColor* method and increments the counter of the enclosing *Point* object. But this may not be intended by the programmer 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 [11]. 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

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

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

or constraint enforcement. But it has been shown that in principle bounded quantification is able to handle also these global concerns as well [14].

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 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: The 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 join points of previous development steps. With ARJ, each development step is represented by a distinct directory, called containment hierarchy [6]. 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.

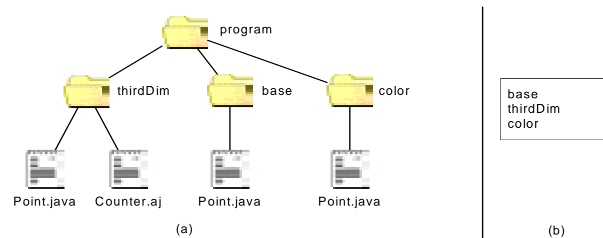

Fig. 4. Program organization. (a) File system, and (b) configuration file.

Having this, the idea of bounding aspect quantification can seamlessly be integrated into ARJ: Since the compiler knows for each aspect to which devel-

opment 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 an aspect-oriented 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 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*. Although its pointcut would originally also match *setColor*, it does not affect the code associated with the third step at all.

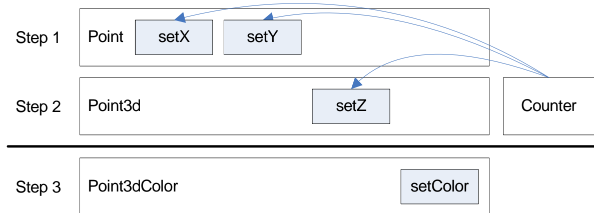

Fig. 5. Incremental weaving approach.

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 this join point with a modified pointcut expression.

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

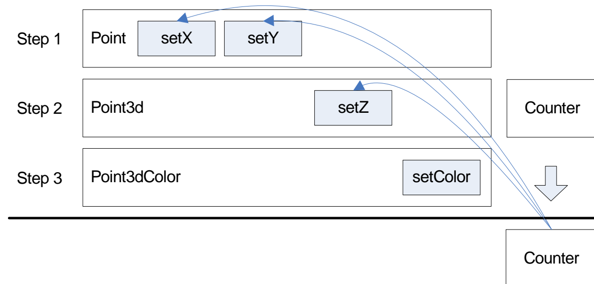

Fig. 6. Pointcut restructuring approach.

uses these annotations to match permitted join points only. For this approach the compiler's frontend and the pointcut matcher must be adapted. Contrasting the restructuring approach, this 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.

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 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. 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. 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 furthermore 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)) || execution(void Point.setY(int)) ||
4      execution(void Point3d.setZ(int))) && target(p);
5   ...
6 }
```

Fig. 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.

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

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

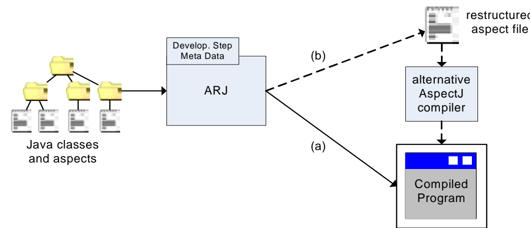

Fig. 9. ARJ compilation process.

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 than be used for debugging purposes or to compile the program with an external AspectJ compiler.

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 but significant 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.

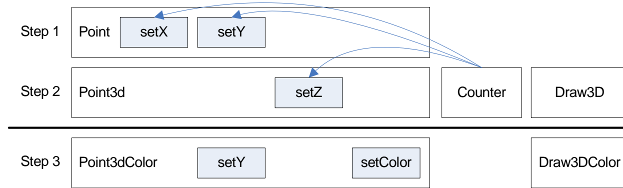

Fig. 10. Extended Example.

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 use to *execution* pointcuts instead, there is no equivalent alternative that matches the access to a field at the callee.

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. An alternative approach is the introduction of a controlled possibility to specify unbounded aspects, as suggested in Section 6.

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 [13, 14, 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 and SA 465/32-1, 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. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6), 2004.
7. A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2004.
8. 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.
9. 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.
10. G. Kiczales et al. Aspect-Oriented Programming. In *Proceedings of European Conference on Object-Oriented Programming*, 1997.
11. R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.
12. D. Lohmann et al. A Quantitative Analysis of Aspects in the OS Kernel. In *Proceedings of ACM SIGOPS EuroSys Conference*, 2006.
13. 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.
14. 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.
15. 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.
16. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2), 1979.
17. A. Tesanovic et al. Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software. *Journal of Embedded Computing*, October 2004.
18. N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4), 1971.
19. 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.