Workshop Proceedings ADI08

F. Sanen M. Sudholt L. Bergmans
R. Chitchyan J. Fabry K. Mehner

Report CW 517, July 2008

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Workshop Proceedings ADI08

F. Sanen *M. Sudholt* *L. Bergmans*
R. Chitchyan *J. Fabry* *K. Mehner*

Report CW 517, July 2008

Department of Computer Science, K.U.Leuven

Abstract

This CW report contains the proceedings of ADI08, the 3rd International Workshop on Aspects, Dependencies and Interactions. The workshop has been organized by the authors of this report: Frans Sanen, Mario Sudholt, Lodewijk Bergmans, Ruzanna Chitchyan, Johan Fabry, Katharina Mehner. It took place on July 8, 2008 in Paphos, Cyprus at ECOOP, the European Conference on Object-Oriented Programming.

Workshop Proceedings

ADI 08

3rd International Workshop on Aspects, Dependencies and Interactions

July 8, 2008, Paphos, Cyprus
In conjunction with ECOOP 2008

Technical Report
Katholieke Universiteit Leuven, No: CW 517
Lancaster University, No: COMP-003-2008
British Library Reference No: ISSN 1477447X

Edited by:

Frans Sanen (K.U. Leuven, Belgium)
Mario Sudholt (INRIA, France)
Lodewijk Bergmans (Universiteit Twente, The Netherlands)
Ruzanna Chitchyan (Lancaster University, UK)
Johan Fabry (Vrije Universiteit Brussel, Belgium)
Katharina Mehner (Siemens, Germany)

Table of contents

Table Of Contents Page 1

Foreword Page 2

Acknowledgements Page 3

Keynote speech Page 4

ADI FULL PAPERS

Management of Aspect Interactions using Statically-Verified Control-Flow Relations
Bruno De Fraine, Viviane Jonckers (Vrije Universiteit Brussel, Belgium), and
Pablo Daniel Quiroga (Universidad Nacional del Comahue, Argentina) Page 5

On Type Restriction of Around Advice and Aspect Interference
Hidehiko Masuhara (University of Tokyo, Japan) Page 15

Towards an Analysis of Layering Violations in Aspect-Oriented Software Architectures
Mario Monteiro, Marcelo Moura, Sergio Soares, and *Fernando Castor Filho*
(University of Pernambuco, Brazil) Page 26

LTS-based Semantics and Property Analysis of Distributed Aspects and Invasive Patterns
Luis Daniel Benavides Navarro, Remi Douence, Angel Nunez, and *Mario Sudholt*
(Ecole des Mines de Nantes, France) Page 36

MDD SUBTRACK PAPERS

Using Transformation-Aspects for Model-Driven Software Product Lines
Hugo Arboleda, Jean-Claude Royer (Ecole des Mines de Nantes, France), and
Rubby Casallas (University of Los Andes, Colombia) Page 46

DiVA: Dynamic Variability in Complex Adaptive Systems
Dhouha Ayed (Thales, France) Page 57

GReCCo: Composing Generic Reusable Concerns
Aram Hovsepyan, Stefan Van Baelen, Yolande Berbers, and *Wouter Joosen*
(K.U. Leuven, Belgium) Page 62

Foreword

Interaction problems between different modules, program parts, units of specifications are a central challenge to many program structuring paradigms, including Aspect-Oriented Software Development, feature-based programming and component-based software engineering. Furthermore, interaction problems are relevant to all phases of the software development life cycle: from requirements through to implementation and often exert a broad influence on these concerns, e.g. by modifying their semantics, structure and / or behaviour. Such dependencies often lead to both desirable and unwanted or unexpected behaviours of large-scale applications.

The goal of the workshop is to continue the wide discussion on aspects, dependencies and interactions, started at ADI 2006 and continued at ADI 2007, thus investigating the lasting nature of such dependency links across all development activities:

- starting from the *early development stages* (i.e. requirements, architecture, and design), looking into dependencies between requirements (e.g. positive or negative contributions between aspectual goals, etc.) and interactions caused by aspects (e.g. quality attributes) in requirements, architecture, and design;
- analysing these dependencies and interactions both through *modelling and formal analysis*;
- considering *language design issues* which help to handle such dependencies and interactions (e.g. 'dominates' mechanism of AspectJ), and, last, but not least
- studying such *interactions in applications*.

It is hoped that input from both research and practice will help to progress the understanding and solutions to this complex subject.

The ADI08 Organizing Committee,

Frans Sanen,
Mario Sudholt,
Ruzanna Chitchyan,
Lodewijk Bergmans,
Johan Fabry,
Katharina Mehner,

July 2008.

Acknowledgements

The Organizing Committee of ADI08 would like to thank the workshop **Program Committee** for their helpful reviews.

- Don Batory, University of Texas at Austin, USA
- Ruzanna Chitchyan, Lancaster University, UK
- Johan Fabry, DCC – University of Chile, Chile
- Alessandro Garcia, Lancaster University, UK
- Iris Groher, Siemens, Germany
- Florian Heidenreich, Dresden University of Technology, Germany
- Andrew Jackson, Trinity College Dublin, Ireland
- Shmuel Katz, Technion, Israel
- Hidehiko Masuhara, University of Tokyo, Japan
- Katharina Mehner, Siemens, Germany
- Klaus Ostermann, University of Aarhus, Denmark
- Frans Sanen, Katholieke Universiteit Leuven, Belgium
- Christa Schwanninger, Siemens, Germany
- Mario Sudholt, INRIA, France
- Jon Whittle, George Mason University at Washington, USA

We are also thankful to **James Noble** for delivering an insightful keynote speech, titled *"We demand rigidly defined areas of doubt and uncertainty!"*, to the workshop.

In addition, we are grateful to **Katharina Mehner** for organizing the panel on *Does model driven engineering make aspects obsolete?* and our panellists.

Finally, we owe a special thanks to **AOSD-Europe** (http://www.aosd-europe.net), the European Network of Excellence on Aspect-Oriented Software Development, for inspiring and supporting this workshop.

Keynote speech

Title: “We demand rigidly defined areas of doubt and uncertainty!”

By: James Noble

Abstract:

A key idea behind aspect-oriented software development is that software cannot be described by tree structures such as OO designs, nested abstractions or layered virtual machines. Unfortunately, this means that the topologies of the software we build, and the interactions within those topologies, will be more complex than we once hoped. This talk will present a philosophical context for this analysis; show how a range of research fits into in that context, and attempt to outline some future directions.

Management of Aspect Interactions using Statically-Verified Control-Flow Relations

Bruno De Fraine[1], Pablo Daniel Quiroga[2], and Viviane Jonckers[1]

[1] System and Software Engineering Lab (SSEL), Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
bdefrain@vub.ac.be,vejoncke@ssel.vub.ac.be
[2] Departamento de Ciencias de la Computación, Universidad Nacional del Comahue, Buenos Aires 1400, Neuquén (Q8300BCX), Patagonia, Argentina
pquiroga@uncoma.edu.ar

Abstract. Although various aspect-oriented approaches provide support for the management of aspect interactions, most techniques are only applicable when the aspects share a common *join point*. However, we observe that aspect interactions also occur on coarser levels, and support for handling these interactions is desirable. In this paper, we demonstrate the feasibility of a technique for managing *control-flow interactions*, one important kind of such interactions that we experience in e.g. layered architectures. The technique proposes to document aspects with *policies* that specify the expected control-flow relations between different aspects, or between aspects and the base application. The policies are expressed as logic formulae that employ a set of predicates that represent relevant control-flow situations. In order to verify the policies, we employ and extend existing static analyses to produce interprocedural control-flow graphs of an application with woven aspects, and we traverse these graphs in a controlled manner to characterize the realizable paths.

1 Introduction

Research in the aspect-oriented community has produced an array of techniques to manage the interactions that may occur between different aspects. On the one hand, a number of approaches support the detection and/or resolution of aspect interaction at a shared join point, independently [1, 2] or in relation to a concrete aspect language or system, such as JAsCo [3], AspectJ [4] or Reflex [5]. The approaches of [6, 7] further augment aspectual advice with documentation about its semantic behavior. This additional information is then used to determine whether advice combinations at a join point entail a semantic conflict. On the other hand, Klaeren *et al.* [8] integrates ideas from feature-based programming and treats interactions at a much coarser level. The authors consider variations of software systems by composing aspects and base classes, and they validate compositions through specified assertions that directly relate these entities.

We believe there is some middle ground between these approaches that has not been explored yet. In this paper, we propose and investigate an integrated

technique to manage control-flow interactions. A typical example of a control-flow interaction is when the application of one aspect changes in one way or another the original control flow and thereby bypasses another aspect's application. In general, control-flow interactions occur between aspects that do not share join points, but are still directly caused by the behavioral changes introduced by them. In a similar way aspect application can interfere with the base program, i.e. a part of the original application can be 'shortcutted'.

1.1 Motivation

We motivate our approach for managing control-flow interactions among aspects with an example scenario in the context of the well-know *multi-tier architectures*, as employed in a large number of current middleware solutions (e.g. JBoss, Spring,...). In these architectures, an application is executed by multiple distinct software agents for reasons of flexibility, scalability, maintainability,... The most widespread case of a three-tier architecture (see figure 1) distinguishes between the presentation tier, the logic tier and the data tier.

In these architectures, each layer will typically consult the underlying layer(s) to complete a request: for example, the presentation tier will delegate a request from the user to the logic tier, which will consult the data tier to retrieve the necessary data. The control flow therefore traverses each of the three tiers, and returns in the opposite direction in order to present the results to the user.

We then consider three typical aspects that might be employed simultaneously in this software system. The first two aspects implement security concerns, while the third aspect implements a performance concern.

1. An **authorization aspect** restricts the access to critical data objects: only configured users (or users assigned to a configured role) can consult or manipulate these data objects. Since the access policies are defined at the level of the data tier, the aspect applies to this tier as well.
2. To enforce security policies, it is necessary to have reliable knowledge about the user that commissioned a certain operation. An **authentication aspect** is therefore used to verify the identity of the user, e.g. by prompting for a user name and password. The aspect applies at the level of the logic tier, because the information is associated with a service request.
3. To speed up the expensive operations of the system, a **caching aspect** is used to store and reuse the results of previous invocations. In order to gain the most benefit, the caching aspect applies directly to the user interface operations in the presentation tier.

While these three aspects apply to different parts of the system (and obviously don't have join points in common), it is certainly possible that they interact in a number of ways with each other, or with the core parts of the application. Below, we identify a number of concrete interactions, and we outline the desired support to manage these interactions.

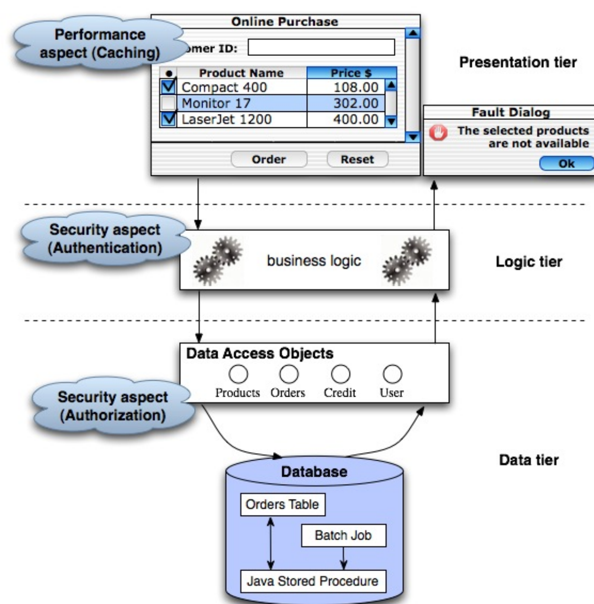

Fig. 1. Security and caching aspects apply at different layers of the architecture

- The authorization concern depends on the presence of authentication in order to correctly enforce access policies. Concretely, we would like to ensure that each application of the authorization aspect is preceded by an application of the authentication aspect.
- Caching should not override the authorization behavior. Since the caching aspect can skip the normal operation by returning a previous result from the cache, the authorization checks are not applied. If a user receives a result that was saved from an invocation by another user, this could bypass his or her access permissions. To avoid this problem, a number of measures are possible (listed in order of increasing sophistication):
  - Verify that the authorization aspect applies on all possible paths starting from important top-level operations.
  - Verify that applications of other aspects do not 'cut off' applications of the authorization aspect (e.g. by comparing the possible paths before and after the weaving of other aspects such as the caching aspect).
  - Set up different cache stores for different users, or otherwise invalidate cache results when they are no longer applicable.

  Whereas the third solution is a very tailor-made resolution that requires global knowledge of the application, the first two solutions only involve the authorization aspect.

1.2 Approach Overview

Because of the dispersed occurrence of the aspects in different layers of a reasonably complex architecture, we anticipate that the manual and/or unsystematic management of these interactions is a complex and error-prone task. We thus identify a need for expressing *control-flow policies* that are able to express control-flow relations (such as "occurs in all paths of" or "cannot remove invocations of") between different aspects or between aspects and other parts of the system. Additionally, it should be possible to write these policies with incomplete knowledge of the system by quantifying over unknown aspects ("any aspect that. . . ") or unknown program parts ("any join point that. . . "). Finally, the policies should be automatically verifiable and should therefore be specified in a well-defined formalism.

In this paper, we propose and investigate an integrated technique to manage control-flow interactions that meets these requirements. This technique consists of three elements that are explained in the following sections: (1) Static analysis of application code with woven aspects, to produce an abstraction of the possible control-flow paths in the resulting application. (2) Formal documentation of aspects with control-flow policies that specify the relations that the aspects depend on. (3) An algorithm to detect violations of control-flow policies in the abstract paths produced as the result of static analysis.

The approach does not (yet) offer guidance to the developer on which policies to document. However, we do envision that policies may be gradually refined when an iterative software development process is used. This means that the aspects may initially be specified with rough policies that may trigger false positives as the system evolves. The developer can then refine the policies with more specific knowledge at the end of each development iteration, based on the results of the automatic policy checking.

2 Static Analysis of Woven Code

We carry out the static analysis of the woven code using an existing Java bytecode analysis tool named Soot [9] (developed by the Sable research group at the McGill University in Montreal). The Soot framework was developed for the purpose of researching optimizing bytecode transformations, but provides a number of analyses that are very useful in our context as well. We first discuss the translations and analyses provided by Soot and then build on them to develop our control-flow analysis of woven code.

2.1 Soot: a Java Optimization Framework

Soot carries out bytecode optimizations by parsing Java bytecode files from various sources to an object-oriented representation. For the representation of method bodies and initializer blocks, the bytecode is additionally translated to four *intermediate representations* that allow to abstract over the technical details

of the bytecode through e.g. generalized instructions, typing of local variables, conversion from stack-based to 3-address code and so on. Optimizations can work at the level of each of these representations, while Soot provides translations between them and the original bytecode form.

Soot can additionally construct a number of graphs describing the program structure. First, *control-flow graphs* of the method bodies and initializer blocks are provided. These graphs are originally used to provide Java decompilation facilities [10] where they serve as a first step to recognize control structures (loops, if-tests,...). The control-flow graph is constructed from a linear representation of the bytecode, by connecting each statement with the statement(s) that might follow it, taking into account (possibly conditional) jumps. Two special nodes are included in each control-flow graph: the *entry node* and the *exit node*, through which all control enters, or respectively leaves, the block.

Second, a whole-program analysis provides a *call graph* with relations between different methods in the application. In this graph, nodes represent the methods in the program, and edges indicate when one method *may* call another. Call graphs are often used to eliminate unneeded methods, or to determine which method bodies may safely be inlined.

The construction of this graph is complicated in an object-oriented context due to polymorphism, since — in principal — all implementations of a method may be selected when the method is invoked on a variable of some supertype. A call graph with this information is produced by *class hierarchy analysis*, but will typically contain a number of edges that cannot appear in the actual application. More precise call graphs are more expensive to calculate, but eliminate these spurious edges, for example, by determining the types instantiated in the entire application (*rapid type analysis*), in the dataflow of the receiving variable (*variable type analysis*) or in the dataflow of all variables of the receiver's type (*declared type analysis*). Using these techniques, Soot is able to provide reasonably precise call graphs that are still practical to compute [11].

2.2 Interprocedural Control-Flow Graphs

The most straightforward solution to combine control-flow graphs of different methods, is to 'inline' the graph of a called method at each call site. The major practical problem with this technique is that the excessive duplication leads to explosive growth of the graph. Worse still, the graph effectively becomes infinite in case of a method that (in)directly calls itself.

The naive technique of connecting the involved control-flow graphs at call sites does not suffer from these complications, but since it cannot distinguish between different calls to the same method, it may introduce *unrealizable paths*. This is illustrated in figure 2: both methods P and R call the method Q, so the call sites are connected to the entry and exit nodes of that method. However, in the resulting control-flow graph, it is possible to enter method Q from P, and leaving it through R, a path not realizable in practice.

To solve this problem, we have to simulate the effect of the invocation stack at the actual execution, and label the entry and exit edges of each invocation

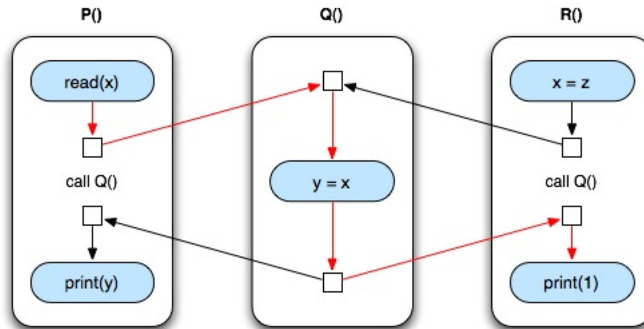

Fig. 2. Unrealizable paths when naively connecting control-flow graphs

with a unique identifier. When traversing the graph, we must maintain a stack with the labels of the edges we have entered, and we can only follow those exit edges whose label is currently at the top of the stack. As such, we avoid following the unrealizable paths that were possible in the naive approach.

2.3 Characterizing Realizable Paths

In order to verify aspect control-flow policies, we need to determine characteristics of the possible maximal (i.e. complete) paths in the interprocedural control-flow graph. Although this graph is finite, there may be infinitely many (and infinitely long) paths due to the presence of cycles in this graph. This renders it complex to decide on properties of these paths.

Fortunately, we observe that it *is* practically possible to calculate a very useful abstraction by characterizing a path as the set of nodes that it encounters[3]. The algorithm to obtain these sets is essentially a straightforward traversal: as we traverse all the realizable paths of the control-flow graph, we propagate the set of encountered nodes and obtain the solutions at the exit node. However, we also record for each node the path(s) (i.e. set(s) of nodes) that passed through the node. When — due to a cycle — we revisit a node, it is only useful to proceed with the current path if it is different from the previous visits of that node (because only in that case we can obtain new paths). By ending the traversal in the other (useless) case, we will guarantee that the algorithm ends[4].

The calculation of these sets is illustrated for a simple graph in figure 3. In the first iteration, we start from the entry node and visit all the nodes a first time, propagating the set of encountered nodes. This already produces two solutions

[3] We mean a mathematical "set" here: an unordered collection without duplicates.

[4] Indeed, since the number of nodes in the graph is finite, the number of sets of nodes is finite as well. So, for each node, it will eventually become impossible to generate a new set that was not previously encountered.

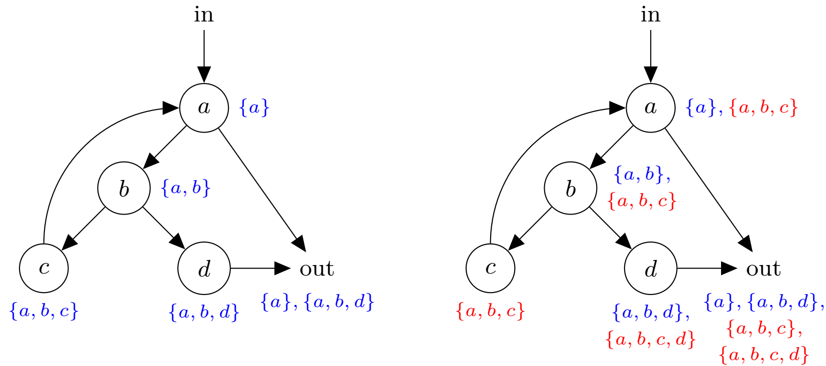

Fig. 3. Illustration of path calculation algorithm in first (l.) and second (r.) iteration

at the exit node. We will then follow the edge from c to a to make a second iteration. We first reach node a with the set $\{a, b, c\}$, which was not considered for this node before, so we can proceed and propagate among the edges leaving node a. We can similarly produce new paths at nodes b and d, but we must halt at node c because the path $\{a, b, c\}$ has already been considered for this node in the previous iteration.

In total, we collect 4 solution sets at the exit node. These sets convey useful information about the (infinitely many) paths that are realizable in practice. For example, they indicate that node a is always applied, and that nodes c or d cannot apply without node b (and vice versa). Since it possible to recognize method nodes and advice method applications (for example, through naming conventions in case of the AspectJ weaver [12]), this analysis provides sufficient information to derive control-flow relations between aspects.

3 Control-Flow Policy Documentation

To specify the control-flow policies that are provided in the aspect documentation, we propose to use a policy language of *first-order logic formulae* (also known as *first-order predicate logic*) [13] that use a number of predefined predicates from the control-flow domain. By combining these atomic propositions with classic logic connectives ($\vee$, $\wedge$ and $\neg$) and quantifiers ($\forall$ and $\exists$), it is possible to build advanced expressions. First-order logic has a number of additional advantages: it is a general and well-understood formal language with well-defined semantics, which makes it suitable for knowledge representation. We can also reuse existing implementation techniques to detect violations of the formulae, provided that we solve the practical problem of integrating the results of the static analysis from the previous section with the reasoning of a logic engine[5].

[5] While we did investigate this issue as well, we do not discuss the results here due to space restrictions (see [14] for details).

3.1 Predicates for the Control-Flow Domain

In order to represent knowledge about the problem domain we define a set of predicates, shown in table 1.

Predicate	Definition
path/2	$path(M, P)$ holds if P is a realizable path from method M
member/2	$member(M, P)$ holds if method M lies on path P
matches/2	$matches(Pa, M)$ holds if method M matches method pattern Pa
adviceof/2	$adviceof(M, A)$ holds if method M is an advice of aspect A
must/2	$must(A, B) \leftrightarrow [\forall P : path(A, P) \rightarrow member(B, P)]$
may/2	$may(A, B) \leftrightarrow [\exists P : path(A, P) \wedge member(B, P)]$
mustnot/2	$mustnot(A, B) \leftrightarrow [\forall P : path(A, P) \rightarrow \neg member(B, P)]$
depend/3	$depend(M, A, B) \leftrightarrow$ $[\forall P : path(M, P) \rightarrow (member(A, P) \rightarrow member(B, P))]$
exclude/3	$exclude(M, A, B) \leftrightarrow$ $[\forall P : path(M, P) \rightarrow \neg\, (member(A, P) \wedge member(B, P))]$

Table 1. Builtin and derived predicates regarding method control-flow relations

The first group of predicates are builtin, and their instantiation follows directly from the analysis of the application code. The *path*/2 predicate provides the results of the calculation of possible complete paths described in section 2.3. Recall that a path is characterized as a set of the encountered method nodes, so *member*/2 is simply the set membership relation. The next two predicates allow to select regular methods and advice methods. As in AspectJ [4], methods can be selected with a method pattern using the predicate *matches*/2. For example, the pattern `* get*(..)` will select all methods with a name that begins with "get", so the predicate *matches* holds for that pattern and the method `Person.getName()` (amongst others). Advice methods are unnamed, and are selected through the type that contains them, using *adviceof*/2.

The second group of predicates is provided as a convenience: they can be derived from the builtin predicates using standard logic formulae, but they allow to specify policies using a slightly higher level of abstraction. These predicates all relate methods. The first three specify the occurrence of the second argument in the control-flow of the first argument: it either appears on all paths (*must*/2), on some path (*may*/2) or on no paths (*mustnot*/2). The next two predicates specify the relative occurrence of the second and third argument in the control-flow of the first argument: the occurrence of one may require (*depend*/3) or exclude (*exclude*/3) the occurrence of the other.

3.2 Example Policies

In section 1.1, we considered the case of an authorization aspect, an authentication aspect and a caching aspect. We observed a number of interactions between

these aspects and we identified a number of policies that we wish to specify for the aspects. We will now express these policies in the policy language.

For the authorization aspect, we wish to specify that it should apply in all paths of a number of important methods. If these methods are selected with the method pattern *main*, we can express that the advices of this aspect *authz* must apply to all realizable paths of the methods matched by this pattern:

$$\forall A, B : \mathit{matches}(\mathit{main}, A) \wedge \mathit{adviceof}(B, \mathit{authz}) \rightarrow \mathit{must}(A, B)$$

We further identified that the authorization logic depends on the presence of the authentication aspect *auth*. We can specify that for all methods matched by pattern *main*, the authorization advice depends on the authentication advice:

$$\forall M, A, B : \mathit{matches}(\mathit{main}, M) \wedge \mathit{adviceof}(A, \mathit{authz}) \wedge \mathit{adviceof}(B, \mathit{auth}) \\ \rightarrow \mathit{depend}(M, A, B)$$

Finally, we specify an alternative for the first policy of this section. The policy to apply authorization in all methods matched by the pattern *main* may be too strict. We can therefore encode a policy that specifies that authorization and caching should be considered exclusive:

$$\forall M, A, B : \mathit{matches}(\mathit{main}, M) \wedge \mathit{adviceof}(A, \mathit{authz}) \wedge \mathit{adviceof}(B, \mathit{caching}) \\ \rightarrow \mathit{exclude}(M, A, B)$$

This works since the caching advice allows two basic paths: one that returns the result from the cache, and another that executes the original behavior. The second path will violate this policy if the original behavior includes the authorization advice.

4 Conclusions and Future Work

In this paper, we propose a technique for managing control-flow interactions, an important kind of interactions that we experienced in e.g. layered architectures. Our approach consists of the documentation of aspects with logic formulae that specify relevant control-flow policies, and the static analysis of the woven application to detect violations of these policies.

As this work explores a new area, we have focused on proving the feasibility of the proposed concepts. Our main additional observation is that the techniques seem useful beyond the anticipated domain of aspect interactions. In general, the automatic verification of aspect control-flow policies can help the programmer enforce design rules in a complex system where it is easy to overlook them. Obviously, this is an issue outside of aspect-oriented contexts as well, and further work should investigate the relationship with general work on the verification of behavioral rules or invariants, such as [15].

References

1. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: 1st Conf. Generative Programming and Component Engineering. Volume 2487 of lncs., Berlin, Springer-Verlag (2002) 173–188
2. Brichau, J., Mens, K., De Volder, K.: Building composable aspect-specific languages with logic metaprogramming. In: 1st Conf. Generative Programming and Component Engineering. Volume 2487 of lncs., Berlin, Springer-Verlag (2002) 110–127
3. Suvée, D., Vanderperren, W.: JAsCo: An aspect-oriented approach tailored for component based software development. In Akşit, M., ed.: Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003), ACM Press (2003) 21–29
4. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: Proc. ECOOP 2001, LNCS 2072, Berlin, Springer-Verlag (2001) 327–353
5. Tanter, E.: Aspects of Composition in the Reflex AOP Kernel. In: Software Composition. Volume 4829 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 98–113
6. Durr, P., Staijen, T., Bergmans, L., Aksit, M.: Reasoning about semantic conflicts between aspects. In Gybels, K., D'Hondt, M., Nagy, I., Douence, R., eds.: 2nd European Interactive Workshop on Aspects in Software (EIWAS'05). (2005)
7. Pawlak, R., Duchien, L., Seinturier, L.: CompAr: Ensuring Safe Around Advice Composition. In: Formal Methods for Open Object-Based Distributed Systems. Volume 3535 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2005) 163–178
8. Klaeren, H., Pulvermueller, E., Rashid, A., Speck, A.: Aspect composition applying the design by contract principle. In: 2nd Int'l Symp. Generative and Component-based Software Engineering (GCSE). Volume 2177 of lncs., Berlin, Springer-Verlag (2000) 57–69
9. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proc. of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. (1999) 125–135
10. Miecznikowski, J., Hendren, L.J.: Decompiling Java using staged encapsulation. In: Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01). (2001) 368–374
11. Sundaresan, V., Hendren, L.J., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00). (2000) 264–280
12. Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In Lieberherr, K., ed.: Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004), ACM Press (2004) 26–35
13. Magnus, P.D.: forall x: An introduction to formal logic, version 1.24. Textbook available at `http://www.fecundity.com/logic/` (2008)
14. Quiroga, P.D.: Control-flow interaction in aspect-oriented programming. Master's thesis, Vrije Universiteit Brussel (2007) In EMOOSE exchange program.
15. Michiels, I.: A Goal-Driven Approach for Documenting and Verifying Design Invariants. PhD thesis, Vrije Universiteit Brussel (2007)

On Type Restriction of Around Advice and Aspect Interference

Hidehiko Masuhara

Graduate School of Arts and Sciences, University of Tokyo
masuhara@acm.org

Abstract. Statically typed AOP languages restrict application of around advice only to the join points that have conforming types. Though the restriction guarantees type safety, it can prohibit application of advice that is useful, yet does not cause runtime type errors. To this problem, we present a novel weaving mechanism, called the *type relaxed weaving*, that allows such advice applications while preserving type safety. This paper discusses language design issues to support the type relaxed weaving in AOP languages.

1 Advice Mechanism in AspectJ

The advice mechanism in aspect-oriented programming (AOP) languages is a powerful means of modifying behavior of a program without changing the program text. AspectJ[6] is one of the most widely-used AOP languages that support advice mechanism. It is, in conjunction with the mechanism called the inter-type declarations, shown to be useful for modularizing crosscutting concerns, such as logging, proflling, persistency and enforcement[1, 3, 10, 12].

One of the unique features of the advice mechanism is the *around advice*, which can change parameter and return values of join points (i.e., speciflc kinds of events during program execution including method calls, constructor calls and fleld accesses). With around advice, it becomes possible to deflne such aspects that directly aflect values passed in the program, including caching results, pooling resources and encrypting parameters. In object-oriented programming, around advice is also useful to modify functions of a system that are represented as objects by inserting proxies and wrappers, and by replacing with objects that ofier difierent functionality.

1.1 Example of Around Advice

We flrst show a typical usage of around advice by taking a code fragment (Fig. 1) in a graphical drawing application[1] that stores graphical data into a flle, which is executed when the user selects the save menu. The hierarchy of the relevant classes is summarized in Fig. 2.

[1] The code fragment is taken from JHotDraw version 6.0b1, but simplified for explanatory purpose.

```
void store(String fileName, Data d) {
  OutputStream s = new FileOutputStream(fileName);
  BufferedOutputStream output = new BufferedOutputStream(s);
  output.write(d.toByteArray());
  output.close();
}
```

Fig. 1. A code fragment that stores graphical data into a file.

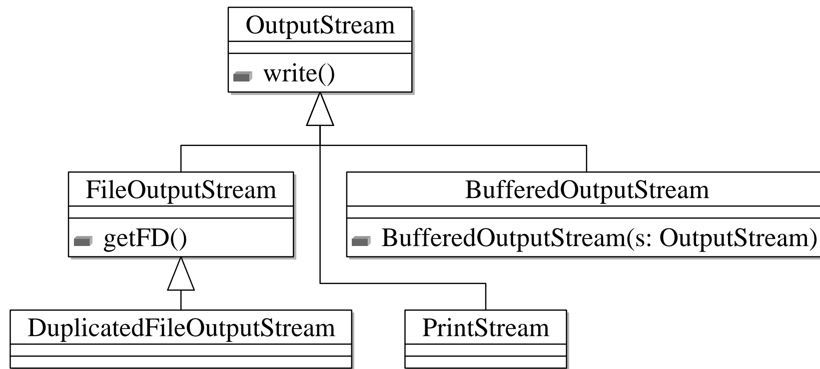

Fig. 2. UML class diagram of the classes that appear in the examples.

Assume we want to duplicate every fle output to the console for debugging purpose. This can be achieved by flrst deflning a subclass of `FileOutputStream` like Fig. 3, and editing the second line of the `store` method (Fig. 1) into the next one.

```
OutputStream s = new DuplicatedFileOutputStream(fileName);
```

Instead of textually editing the `store` method, we can deflne an aspect that creates `DuplicatedFileOutputStream` objects instead of `FileOutputStream`, as shown in Fig. 4. By using the aspect, we can replace creations of objects that are performed in many places in the program. In fact, there are several classes in JHotDraw that create `FileOutputStream` in order to support difierent fle formats. The aspect deflnition can therefore achieve more modular modiflcation.

2 The Problem: Restriction on Types of Around Advice

2.1 Restrictions for Type Safety

AspectJ guarantees type safety of around advice by type checking the body of around advice with respect to its parameter and return types, and by checking the return type of the around advice with respect to the return types of join points which the advice is woven into.

```
class DuplicatedFileOutputStream extends FileOutputStream {
  DuplicatedFileOutputStream(String filename) { super(filename); }
  void write(int b) { super.write(b); System.out.write(b); }
  ...override other methods that write data...
}
```

Fig. 3. A class that duplicates all file outputs to the console.

```
aspect Duplication {
  FileOutputStream around(String n):
      call(FileOutputStream.new(String)) && args(n) {
    return new DuplicatedFileOutputStream(n);
  }
}
```

Fig. 4. An aspect that replaces all creations of `FileOutputStream` with `DuplicatedFileOutputStream`.

The type checking rules can be summarized as follows. Given an around advice declaration with return type `T`, pointcut `P`, and a return statement with expression `e`:

```
T around(): P { ... return e; ... }
```

the program must satisfles the next two rules.

(AJ1) The type of `e` must be a subtype[2] of `T`.
(AJ2) For each join point matching `P`, `T` must be a subtype of its return type.

For example, the next around advice is rejected by **AJ1** because the type of `System.out`, namely `PrintStream`, is not a subtype of `FileOutputStream`.

```
FileOutputStream around(): call(FileOutputStream.new(String)) {
    return System.out; //of PrintStream --- type error
}
```

The next advice declaration is rejected by **AJ2**. The pointcut of the advice matches a join point that corresponds to an expression `new FileOutputStream(...)` in `store` (Fig. 1). Since `String` is not a supertype of `FileOutputStream`, the advice violates **AJ2**.

```
String around():                                //weaving error
    call(FileOutputStream.new(String)) {
  return "Hello!";
}
```

[2] Here, we assume the subtype and supertype relations are reflexive; i.e., For any `T`, `T` is a subtype and supertype of `T`.

```
aspect Redirection {
  PrintStream around(): call(FileOutputStream.new(String)) {
      return System.out; // of type PrintStream
  }
}
```

Fig. 5. An aspect that returns the `PrintStream` object (i.e., the console) instead of creating `FileOutputStream`. While current AspectJ compilers reject this aspect as type incompatible at weaving, our type relaxed weaving accepts this.

2.2 The Problem: Useful, yet Prohibited Advice

We found that the restrictions prohibit to define some useful advice. Assume we want to redirect the output to the console, instead of duplicating. This can be easily achieved by editing the second line of `store` (Fig. 1) into the next one.

```
OutputStream s = System.out; //of type PrintStream <: OutputStream
```

This is type safe, because the field `System.out` is of type `PrintStream`, which is a subtype of `OutputStream`.

However, it is not possible to do the same replacement by using AspectJ's around advice. The aspect declaration in Fig. 5 seems to work at first glance, but is actually rejected by **AJ2**. This is because **AJ2** requires the return type of the matching join point (`FileOutputStream`) to be a supertype of the return type of the advice (`PrintStream`). Changing the return type of the advice into `FileOutputStream` does not work, because it will violate **AJ1**.

2.3 Most Speci c Usage Type

The problem can be clarified by using the notion of *the most specific usage type* of a value. We define the *usage types* of a value as follows. When a value is used as a parameter or a receiver of a method or constructor, the usage type of the value is its static parameter or receiver type, respectively. When a value is returned from a method, the usage type is the return type of the method. The most specific usage type of a value is such `T` that `T` is a subtype of any usage type of the value, and when `T'` is a subtype of any usage type of the value, `T'` is a subtype of `T`.

For example, the return value from `new FileOutputStream(..)` in `store` (Fig. 1) is used only as a parameter to the `BufferedOutputStream` constructor. Therefore, its most specific usage type is the static parameter type of the constructor, namely `OutputStream`.

The most specific usage type of an expression's return value indicates the upper bound of return types of replacement expression. In other words, replacing the expression with another expression succeeds when the return type of the replacement expression is a subtype of the most specific usage type. In the `store`'s case, the `new FileOutputStream(..)` expression can be replaced with any expression that has a subtype of `OutputStream`.

By using the most specific usage type, the problem in the previous section can be stated as follows:

Problem. AspectJ prohibits an around advice declaration when its return type `T` *is not a subtype of the return type of a matching join point, even if* `T` *is a subtype of the most specific usage type of the return value of the join point.*

2.4 More Example of the Problem

The problem is not artificially crafted. It rather can be found more frequently. Another example is around advice that wraps handlers. Java programs frequently use anonymous class objects in order to define event handlers. For example, the next code fragment creates a button object, and then installs a listener object into the button object. The listener object belongs to an anonymous class that implements the `ActionListener` interface. The parameter type of the `addActionListener` method is `ActionListener`.

```
JButton b = new JButton();
b.addActionListener(
  new ActionListener () {
    public void actionPerformed(ActionEvent e) {...}
  }
);
```

Now, assume that we want to wrap the listener object with an object of `Wrapper` that also implements `ActionListener`. While textually inserting a constructor call around the `new` expression is type safe, the next aspect that does the same insertion violates **AJ2**.

```
aspect WrapActionListener {
  ActionListener around(): call(ActionListener+.new(..)) {
    ActionListener l = proceed();
    return new Wrapper(l); // Wrapper implements ActionListener
  }
}
```

Note that the pointcut captures any construction of objects that are of a subtype of `ActionListener` (cf. the plus sign after the class name means any subtype of the class).

2.5 Generality of the Problem

While the examples are about return types of around advice in AspectJ, the problem and proposed solution in the paper are not limited to them.

First, the problem is not limited to the return type of around advice. Since around advice can also replace values of parameters that are captured by `args`

and `target` pointcuts, the same problem arises at replacing those values by using the `proceed` mechanism.

Second, the problem is not limited to AspectJ. Statically-typed AOP languages that support around advice, such as CaeserJ[9] and AspectC++[11], should also have the same problems.

Third, the problem is not limited to AOP languages. The same problem would arise the language mechanisms that can intercept and replace values, such as method-call interception[7] and type-safe update programming[4].

3 Type Relaxed Weaving

3.1 Basic Idea

We propose a weaving mechanism, called *type relaxed weaving*, that solves the problem. The only difierence from the original AspectJ is **AJ2**, whose new rule is shown below.

(AJ2′) For each join point matching `P`, `T` must be a subtype of its *most specific usage type* of its return value.

The type relaxed weaving accepts the aspect `Redirection` in Fig. 5 and `WrapActionListener` in Section 2.4. For the `Redirection`'s case, the return type of the advice (`PrintStream`) is a subtype of the most speciflc usage type of the join point (`OutputStream`). For the `WrapActionListener`'s case, the return type of the advice (`ActionListener`) is a subtype of the most speciflc usage type of the join point (`ActionListener`).

3.2 Design Issues

There are several design issues that need to be addressed in order to make the type relaxed weaving into a concrete language implementation.

Operations that use values. We deflne the next eight operations *use* a value. (1) Calling a method or a constructor with the value as a parameter. (2) Calling a method with the value as a target. (3) Returning from a method with the value. (4) Down-casting the value. (5) Accessing (i.e., either reading from or writing to) a fleld of the value. (6) Assigning the value into a fleld. (7) Assigning the value into an array. (8) Throwing an exception with the value.

Note that the operations do not include assignments to local variables. This is because, types of local variables may not be available at bytecode weaving. Also, excluding local variable assignments can give more opportunities to type relaxed weaving. For example, when local variable assignments are *not* considered as use of values, the most speciflc usage type of the constructor call to `FileOutputStream` in the next code fragment is `OutputStream`, which is otherwise `FileOutputStream`.

```
FileOutputStream s = new FileOutputStream(fileName);
BufferedOutputStream output = new BufferedOutputStream(s);
```

Usage type of an overridden method call. When a value is used as a target object of a method call, we regard it as a usage with type `T` where `T` is the most general supertype of the static type of the value that defines the method.

Note the usage type of a target object can be different from the target type in the method signature that the Java compiler gives. For example, to the method call `output.write(...)` in `store` (Fig. 1), recent Java compilers chooses `BufferedOutputStream.write(byte[])` as its method signature because `BufferedOutputStream` is the static type of `output`.

The usage type of the value of `output` in the call is, however, `OutputStream` because it is the most general type that defines `write`.

Extent of value tracking. When computing the most specific usage type of a value, we merely chase dataflow of values within a method. This is a connotation of our previous decision that regards parameter passing as the usage of a value.

There could be further type relaxation opportunities if we took dataflow across method calls into account. Assume that the return value from `new FileOutputStream(..)` is passed as a parameter to the constructor `BufferedOutputStream(OutputStream)` (as shown in Fig. 1), but the constructor used the parameter as a value of type `Object`. Then, it is theoretically safe to replace the value with the one of any subtype of `Object`.

We however did not choose this level of relaxation because its implementation requires inter-method dataflow analysis as well as changes of method signatures, which are not easy.

When a usage type does not match. There are two options for the weaver when it detects that the return type of around advice is not a subtype of the usage type of a matching join point. The one is to raise an error as the current AspectJ compilers do. The other is not to weave the advice at the join point; i.e., we will weave advice to only join points that have more general usage types than the return type of the advice. Our tentative choice is the former. We believe further experiences will reveal the advantages and disadvantages of those options.

3.3 Aspect Interaction

When more than one aspect interacts, i.e., more than one around advice declaration is applicable to one join point, care must be taken about type safety.

Assume there are two around advice declarations: the one is in the `Redirection` aspect (Fig. 5), and the other is as follows.

```
FileOutputStream around(): call(FileOutputStream.new(String)) {
  FileOutputStream s = proceed();
  ... s.getFD() ... // getFD() is defined only in FileOutputStream
  return s;
}
```

Note that the advice uses the return value from `proceed` as the one of type `FileOutputStream`.

When those advice declarations are applied to a constructor call to `FileOutputStream` (e.g., the second line of `store` in Fig. 1), it is not safe depending on the advice precedence. If the latter advice precedes the former, the latter receives a value that is replaced by the former: i.e., a `PrintStream` object. The method call at the third line in the latter advice declaration then fails because the method is deflned only in `FileOutputStream`. If the former precedes the latter, there is no problem because the former does not proceed further.

To cope with this interaction problem, we need to take the usage in advice bodies into account. Given a join point, we regard that its return value is used by advice bodies in addition to the operations listed in Section 3.2. In the above example, the latter advice uses the return value from `proceed` as `FileOutputStream`. Therefore, the most speciflc usage type of the join point becomes `FileOutputStream`, which correctly detects the problem of combining those two advice declarations.

3.4 Implementation

We are implementing a system that allows the type relaxed weaving. The implementation is based on existing AspectJ compilers extended with a modifled weaving algorithm. Since the difierence is only in between **AJ2** and **AJ2′**, the modiflcation should be minimal.

The implementation consists of two parts, namely an analyzer that computes the most speciflc usage type of a given expression, and a judgment routine that decides whether a weaver can apply an around advice body to a join point.

The analyzer can be implemented as a bytecode verifler for the Java bytecode language that checks type safety of a method given in a bytecode format. The difierence from standard bytecode verifler is that it performs type inference with the return type of a specifled method call unknown. Since we decided to track dataflow merely within a method, it is not di– cult to implement the analysis.

The judgment routine is invoked when a weaver is to apply a set of applicable advice declarations to a join point. When the set does not include around advice, it proceeds as usual. When it does, it flrst computes the most speciflc usage type of the join point, as well as the most speciflc usage type of the body of each applicable advice. It then flnds the greatest type `T` that is a subtype of all the most speciflc usage types. Finally, it allows advice weaving when the return type of the around advice is a subtype of `T`.

4 Preliminary Feasibility Assessment

Before implementing the system, we estimated the number of opportunities in practical programs that can beneflt from the type relaxed weaving. We monitored executions of flve medium-sized Java programs, and classifled the join point

shadows (i.e., source code locations) based on their most speciflc usage types. We used AspectJ for monitoring program executions.

The evaluated programs and the results are summarized in Table 1. The 'more general' row shows the numbers of shadows whose most speciflc usage type are strict supertypes of the return types or parameter types of the shadows. They approximate the numbers of shadows that can beneflt from the type relaxed weaving. As can be seen in the table, we found that approximately 15 30% are such shadows. Even though the numbers merely represent potential beneflts, they suggest that the type relaxed weaving can be useful in practice.

5 Related Work

There are several researchers that work on the types of the advice mechanism in AspectJ-like languages.

Clifton and Leavens formalized the proceed mechanism of around advice and its type safety[2]. They directly work on the type system of around advice, but for formalizing the existing mechanism. Our work purposes to deregulate existing mechanism for enabling more useful around advice declarations.

StrongAspectJ oflers an extended mechanism and a type system for around advice in AspectJ[5]. The purpose of the work is to support generic around advice without compromising type safety. It would be useful to deflne an around advice declaration applicable to many diflerent join points with diflerent types, as long as the advice body does not access the values obtained from join points. We believe our proposed mechanism complements their mechanism, as ours aims at supporting advice declarations that explicitly changes types of values exchanged between advice and join points.

Aspectual Caml is an AOP extension to the functional language Caml[8] that supports polymorphic pointcuts. Similar to StrongAspectJ, polymorphic pointcuts enable to weave advice declarations into join points with diflerent types as long as the advice body does not access the values in join points.

Table 1. Characteristics of the measured programs and the results. The bottom four rows show the numbers (and their percentages in parentheses) of join point shadows whose return values or parameter values are used as the values of more general, more specific, incompatible or same types of the shadows.

program name	Javassist	ANTLR	JHotDraw	jEdit	Xerces
program size (KLoC)	43	77	71	140	205
number of shadows	862	1,827	3,558	8,524	3,490
more general(%)	177 (21)	315(17)	576 (16)	2,499(29)	650(19)
more specific(%)	37 (4)	70 (4)	170 (5)	974(11)	156 (4)
incompatible(%)	0 (0)	4 (0)	42 (1)	42 (0)	64 (2)
same(%)	648 (75)	1,438(79)	2,770 (78)	5,009(59)	2,620(75)

6 Conclusion

This paper presented a problem of type restriction to around advice that prevents the programmer defining advice that replaces values into of different types, while the replacement is type safe when it is achieved by textual modification. To the problem, we proposed the notion of *the most specific usage type* and a novel weaving mechanism called the *type relaxed weaving*, which permits around advice to replace values with the ones of the most specific usage type. Our preliminary assessment showed that practical programs have 15 30% join points (per source code location basis) that can benefit from the type relaxed weaving.

We are currently implementing an extended AspectJ compiler that supports type relaxed weaving. The implementation will consist of the type inference algorithm in Java bytecode verifier, and a small modification to the weaving mechanism in an existing AspectJ compiler.

We also plan to formalize the mechanism so that we can prove type safety with the type relaxed weaving. We believe this is needed for ensuring the correctness of the algorithm in complicated cases, especially when more than one aspect interacts.

Acknowledgments

The author would like to thank Atsushi Igarashi, Tomoyuki Aotani, Eijiro Sumii, Manabu Touyama, the members of the PPP research group at University of Tokyo, the members of the Kumiki 2.0 project and the anonymous reviewers for their helpful comments.

References

1. Ron Bodkin. Performance monitoring with AspectJ: A look inside the Glassbox inspector with AspectJ and JMX. AOP@Work, September 2005.
2. Curtis Clifton and Gary T. Leavens. MiniMAO1: Investigating the semantics of proceed. *Science of Computer Programming*, 63(3):321–374, December 2006.
3. Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 56–65, March 2004.
4. Martin Erwig and Deling Ren. Type-safe update programming. In *ESOP 2003*, volume 2618 of *Lecture Notes in Computer Science*, pages 269–283, 2003.
5. Bruno De Fraine, Mario Südholt, and Viviane Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 60–71, April 2008.
6. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. June 2001.

7. Ralf Lämmel. A semantical approach to method-call interception. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 41–55. April 2002.
8. Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proceedings of International Conference on Functional Programming (ICFP 2005)*, pages 320–330, September 2005.
9. Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. March 2003.
10. Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD2003)*, pages 120–129. March 2003.
11. Olaf Spinczyk, Andreas Gal, and Wolfgang Schroder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 18–21, February 2002.
12. Daniel Wiese, Regine Meunier, and Uwe Hohenstein. How to convince industry of AOP. In *Proceedings of Industry Track at AOSD.07*, March 2007.

Towards an Analysis of Layering Violations in Aspect-Oriented Software Architectures

Mário Monteiro[1], Marcelo Moura[2], Sérgio Soares[3], Fernando Castor Filho[4]
Department of Computing and Systems, University of Pernambuco
Rua Benfica, 455, Madalena, 50.750-410, Recife, Pernambuco, Brazil
{mqm, mlmm, sergio, fernando.castor}@dsc.upe.br

Abstract. Empirical experiments with quantitative results are of great importance to analyze the benefits and drawbacks of aspect-oriented programming (AOP) in different contexts. Although many assessments of this new paradigm have been conducted in the last few years, only a small number of studies address specifically the impact of AOP on the architecture of a software system. This seriously hinders the adoption of AOP, as AOP techniques challenge the traditional notion of modularity on which most of the research on software architecture is based. More specifically, it is not clear what the effect of AOP on layered software architectures is. In this work, we present a study with the goal of analyzing the influence of AOP on violations of the layered structure of software architectures. We argue that the existing metrics for layering violations do not appropriately accommodate the notion of aspects. We explain the problems with these metrics and motivate the need to extend them to allow more precise quantitative evaluations of layering violations when aspects are involved. The target application of the study is a real-life web-based system that has been used in many other scientific studies.

Keywords: aspects, software architecture, layered software architectures, software evolution.

1 Introduction

Aspect-Oriented Programming (AOP) [9] is a software development approach that supports the modularization of crosscutting concerns in complex software systems. Although many assessments of AOP have been conducted in the last few years [15] [6] [12] only a small number of studies addresses specifically its impact on the architecture of a software system. This seriously hinders the adoption of AOP, as AOP techniques challenge the traditional notion of modularity on which most of the research on software architecture is based [1]. More specifically, it is not clear what the effect of AOP on layered software architectures is.

Layered software architectures were introduced by Djikstra [5] in the context of operating systems research. A layered system is structured as hierarchy of modules where each module (or *layer*) only can use services that the module located immediately below it offers. Layered architectures promote benefits such as maintainability and the ability to separately understand the parts of the system. Since

[1] Supported by FACEPE.
[2] Supported by CAPES.
[3] Partially supported by CNPq, grant #480489/2007-6.
[4] Partially supported by CNPq, grants #481147/2007-1 and #550895/2007-8.

their inception, layered software architectures have been widely adopted and there are many existing systems that are structured as sets of layers. Notable examples include most of the existing web-based information systems and network protocol stacks such as the Internet's [13].

In this work, we study the impact of AOP techniques on layered software architectures. We have conducted an empirical study targeting five evolution scenarios of a real-life web-based information system to better understand this impact. More specifically, we focus our analysis on layering violations, that is, situations where a layer is a client to another layer that is not below it or not adjacent to it. Although there are already studies on layering violations in the literature [17], to the best of our knowledge, none of them gives specific attention to aspects. Actually the study [17] is an important related work, since it provides a metric suit to quantitatively measure layering violations in source code. The contributions of this work are the following:

- A motivation for the need to adapt existing layering violation metrics to take aspects into account.
- Development of a framework, using the AspectJ language, to automatically measure layering violations in Java programs.
- Quantitative and qualitative evaluation of a software system in terms of layering violations. In this study, we consider five different evolution scenarios of the same system and employ two implementation approaches: Object-Oriented Programming (OOP) and AOP.

This paper is organized as follows. Section 2 introduces the setting of this study, whereas Section 3 presents its target application, Health Watcher [18]. Section 4 describes the notion of layer that we adopt in this work. This is necessary because architectural modules are often not explicitly materialized at the implementation level. Section 5 presents a framework that we have developed in order to collect information about dependencies between layers. Finally, Section 6 presents and analyzes the results of our study and Section 7 rounds the paper.

2 Study Setting

This study is divided into five major phases: (1) selection of a target application, including the selection of relevant evolution scenarios; (2) documentation and placement of each module in a specific layer; (3) analysis and motivation for the need to adapt existing layering violation metrics, so as to consider the effects of AOP; (4) development of a framework to automatically collect the metrics; (5) evaluation, both quantitative and qualitative, of layering violations in the target application, considering five evolution scenarios and two versions, one aspect-oriented and the other one object-oriented. Sections 3-7 describe each of these phases, respectively.

In order to analyze architectural layering violations in the web information systems domain, this work considers two different implementations techniques: AOP and OOP. We chose AspectJ [10] as a representative of AOP approaches, since it is the most mature and widely adopted amongst existing AOP languages. It is important to stress that our goal is not to compare different AOP mechanisms. Therefore, we attempt to be consistent in our use of AOP mechanisms and always opt for the simplest possible solution to solve design and implementation problems. Our main goal is to assess the impact that AOP technology has in a layered software

architecture, in terms of creating, removing, and/or changing layering violations. As a representative of OOP, we employ Java because it is often used to implement web-based information systems and it is the language from which AspectJ derives.

The quantitative metrics to determine layering violations [17] are very important in our study. They are the main tool we employ to analyze the influence that AOP and OOP have in terms of layering violations. We consider three types of layering violation: Skip-Call, Back-Call and Cyclic Dependency. We explain each type of violation using an example. Let *A* and *B* be two components in a software architecture. Assume that *A* invokes services from *B* and that *A* and *B* belong to layers *LA* and *LB*, respectively. In this scenario, we define the types of layering violations as follows:

- *Skip-Call*: Occurs if *LB* is located below *LA*, but they are not adjacent, i.e., there is some other layer, *LC,* between *LA* and *LB*.
- *Back-Call*: Occurs if *LB* is located above *LA*.
- *Cyclic Dependency*: Occurs if there is a cycle in the dependence graph formed by the dependencies amongst the layers of the system.

More details about these types of layering violations can be found elsewhere [17].

3 Target Selection

The first important decision in this study is the selection of the target application. We have chosen a typical web-based information system, called Health Watcher (HW) [18]. HW is a complaint registration system. Its major objective is to improve the quality of the services provided by the health care institutions, by allowing the public to register complaints about the quality of the service provided to them. In this study, we did not consider another target application, but we believe that more studies should be taken into account in order to collect more evidences considering our approach (see Section 7).

This system was selected due to the several reasons related to our study goals and constraints. First, it is a real and non-trivial application that has a pure Java version and an AspectJ version where some concerns, most notably distribution and persistence, are modularized through aspects. This factor can help us to better identify the effect that each implementation technology has on the study results. The HW design presents common quality requirements, e.g. web-based GUI, persistence, concurrency, distribution, and implementation technologies, e.g. Servlets, JDBC, and RMI. Besides, it has a good representativeness in terms of crosscutting and non-crosscutting concerns [8], i.e., there are a variety of real-life application concerns. This justifies the aspect-oriented implementation besides the object-oriented one.

Second, HW was developed based on classical n-tier architecture, a very well-known and widely used layered architecture. Therefore, we believe it is a good representative of real layered software systems.

Finally, one of the previous empirical studies targeting HW [8] applied a number of maintenance and evolution change scenarios to a baseline version of the system. This effort aimed to simulate a software development environment with realistic change scenarios. In this study, we selected the first five among these scenarios, as depicted in Table 1. Each scenario was applied to two different branches of HW: one purely object-oriented and the other one using AOP. Hence, each release of the system produces two different versions.

Table 1. HW evolution scenarios employed in this study.

Release	Scenario description
R1	Factor out multiple Servlets to improve extensibility.
R2	Ensure the complaint state cannot be updated once closed to protect complaints from multiple updates.
R3	Encapsulate update operations to improve maintainability using common software engineering practices.
R4	Improve the encapsulation of the distribution concern for better reuse and customization.
R5	Generalize the persistence mechanism to improve reuse and extensibility.

4 Identifying and Documenting Architectural Layers

Before we start measuring architecture layering violations, it is necessary to identify what are the layers in the system and to which layer each module belongs. First, we consider that each implementation unit in the source code is a module , i.e., a module can be a class, an aspect, or an interface.

Every module should belong to a certain layer [17]. Besides, considering a Layer L, argument, return, and error types of operations provided by modules on the border of L (invoked by modules from other layers) must be defined either in L or, more commonly, in shared data definition modules [2]. However, in order to measure architectural layering violations, we need to place shared data definition modules into a certain layer. Sometimes it is difficult to determine when to assign a module to a certain System layer. It is clear that a module responsible for persisting objects should be in the data layer, a module that manages user interaction, appearance, button events and menus should be in the graphical user interface (GUI) layer. However, it is not clear in which layer reusable elements on which modules located in many different layers of the system should be placed. Examples of such elements would be the `Account` class in a banking system or, in the case of HW, the `Complaint` class. We view these classes as offering services to modules located in many other layers of the system, even though they do not offer services in strict sense. Reusable classes usually do not require services from GUI nor business modules. Instead, basic classes are used by all of them. This line of reasoning led us to place these elements at the lowest layer of the hierarchy, which we shall call "Reusable Elements layer".

The design choice of placing reusable elements in the lowest architectural layer is based on the assumption that skip calls are a less severe layering violation than back calls. This is intuitive if we consider that one of the motivations for the use of layered architectures is to be able to replace a certain layer by another one without affecting any lower layers. When a module makes a back call, it breaks this principle and, as a consequence, partially defeats one of the main benefits of layered architectures. If, on the other hand, a module at a layer *L* makes a skip call, this means that it depends on layers that are not adjacent to *L*, but are still below it. Hence, *L* can still be replaced without affecting layers located at lower levels. By placing reusable elements at the lowest layer, we avoid making back calls to them by introducing some skip calls. The UML Components software development method [3] takes a similar approach to separate reusable from non-reusable software components.

Another problem arises in the aspect-oriented systems domain. Aspects are modules and, therefore, should belong to a certain layer [17]. However, it is not clear to which layers the aspects belong. There are examples of crosscutting that fit appropriately within a layered structure, e.g. Distribution. Nevertheless, for simplicity, since this is still an ongoing work and most of the crosscutting concerns of Health Watcher cannot be matched to a specific layer, we argue that aspects do not belong to any of the layers. In fact, aspects that are intended to crosscut can easily cross layers, so arbitrarily assigning them to a layer obviously invites problems. Therefore, it is not straightforward to claim that aspects should be localized into layers. In this case, we have to treat the metrics defined in [17] differently when dealing with aspect-oriented systems.

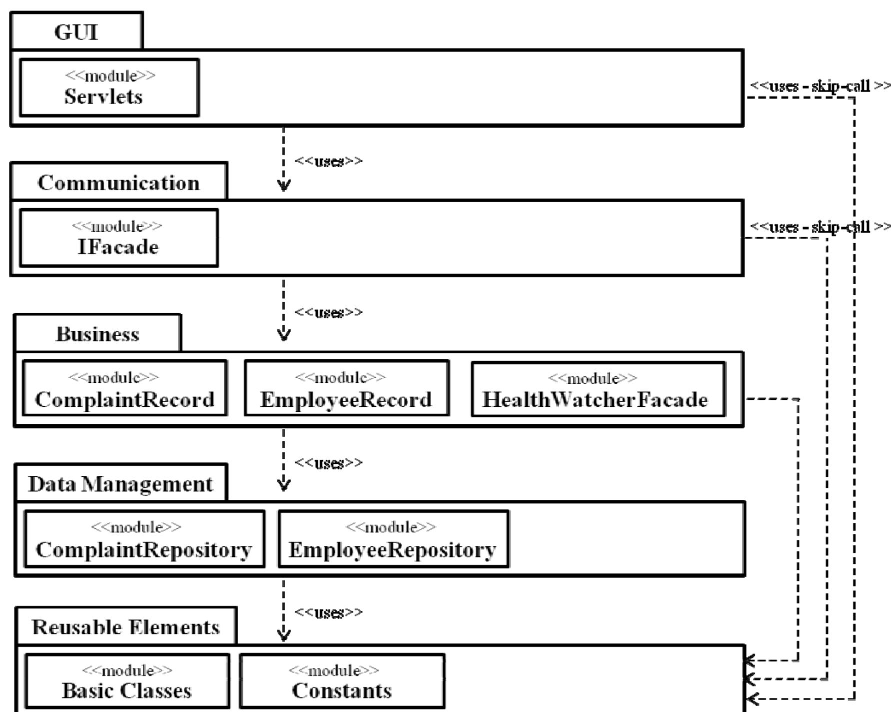

Figure 1. HW layer architecture.

The existing metrics for architectural layering violation [17] do not consider aspects. However, even though we believe, in this study, that aspects should not belong to any layers, there can still exist many dependencies from aspects to modules within the layers. This raises the question of whether aspects should be considered, for the purpose of counting skip-call, back-calls, and cycles. Our opinion is that aspects should not be considered for this purpose. These types of violation are inherently dependent on the location of a layer in the architectural layers hierarchy and aspects, considering Health Watcher System, are not part of any specific layer. It is undeniable that AOP still affects the results of our study, since their introduction changes some dependencies in the system, as discussed in Section 6. Nevertheless, ideally, the influence of aspects in layering violations should also be measured directly, in terms of dependencies on/from aspects, and not only indirectly, in terms of the modules that aspectization affects. This kind of measurement is important for

maintenance because changing a module on which an aspect depends might break the aspect. Currently, though, there is no metric suite that bridges the gap between a layered architecture and the crosscutting dependencies that AOP makes explicit and we consider this fertile ground for future research. The last section of this paper briefly elaborates on this topic.

In order to ease the identification and documentation process, we have developed a simple tool to assist us in associating modules to layers. According to the HW System documentation [8], the modules can be placed in four layers: (1) GUI, (2) Communication, (3) Business, (4) Data Management. As previously mentioned, we have also defined a fifth layer: the Reusable Elements layer. Every aspect implementation unit is registered so that it does not belong to any layer. After the registration of all modules, a configuration file is created with all registered information and then it is used as input by the framework (explained in detail in Section 5), in order to automatically collect the layering violation metrics.

All module registration was carried out carefully, with professional supervision (HW developers) and researchers with long-term experience on the development of the HW. As much as possible, we have employed automated tools to reduce the impact of human errors in the registration process. The correctness of this task is primordial, since it will directly affect all metrics.

Figure 1 depicts the HW architecture. Due to space constraints the figure shows only a few modules and examples of skip-calls violations.

5 Architectural Layer Violation Measurement Framework

After registering all implementation units mentioned in Section 4, we need to identify all the dependencies between modules. In this work, we adhere to the definition of dependency adopted by the UML [16]. The latter considers that a dependency between two elements implies that if one of them changes, the other one might have to change as well. For simplicity, we consider only the following five kinds of dependencies:

1. **Method calls**: Module A depends on Module B if A calls a method from B, or if A instantiates B.
2. **Field Access**. Module A depends on Module B if A reads some field from B.
3. **Field Assignment**: Module A depends on Module B if A performs an assignment to a field from B.
4. **Exception Handling**. Module A depends on Module B if A handles exception.

All these dependencies should be considered in the entire system. The identification of such dependencies should not be done manually, since this is an error prone and time-consuming task. Therefore, we developed a simple framework using the features available in AspectJ to identify all the dependencies automatically.

The framework provides two major functionalities: (1) identification of dependencies; and (2) metrics collection.

Identification of Dependencies. This functionality is responsible for determining all dependencies between modules and for creating a text file with this information. First of all, it is necessary to create a pointcut selecting all joint points that represent a

dependency on a given module (in this example, module Address), along with a declare error statement as shown in the code snippet below.

```
pointcut methCallAddress():call(* Address.*(..)) && !within(Address);
pointcut constructorAddress():call(Address.new(..))&& !within(Address);
pointcut getValueAddress():get( * Address.*) && !within(Address);
pointcut setValueAddress():set( * Address.*) && !within(Address);
pointcut handlerAddress():handler(Address) && !within(Address);
pointcut dependencies():methodCallAddress() || constructorCallAddress() ||
                        readFieldValueAddress() || assignFieldAddress()  ||
                        handleAddress();
declare error : dependencies() : "#Address#";
```

The declare error statement will cause a compilation error whenever a module (that is not Address itself) depends on it. Therefore, the compilation error log contains all dependencies from all modules to the module Address. If similar code is produced for every module of the system, the error log will contain all the dependencies for a specific project. The construction of the code snippet above can be automated for all modules from a project, since the only difference is the name of the module in each pointcut and declare error statement. In our study, we developed a small program which accesses each module from a given project in the source code folder and then generates dependency-tracking aspects for all implementation units. The result is set of aspects including a declare error statement for each module of the project. We treat the error log as the dependency graph and use it as input to metrics collection. In this study, the AspectJ error log is generated in the Problem View (using Eclipse/AJDT) and the contents (which is very simple to read and understand) is saved as a simple text file, where each line represents a dependency. In this case, a simple text parser is sufficient to obtain the graph dependency automatically.

Metrics Collection. This phases takes as input the list of layers in the system, including their order, set of modules associated to each layer (Section 4), and the dependency graph, and uses them to detect layering violations. The framework works by first picking a dependency of an arbitrary module A on a module B and then determining to which layer each of these modules depend. It then checks, for each dependency from a module A on a module B, whether: (i) A's layer is higher than B's; and (ii) A and B are adjacent. The first condition detects back calls, whereas the second one detects skip calls. The framework detects Cycles in the same way that cycles are usually detected in graphs [4]. This procedure is repeated for every dependency in the dependency graph and the result is a text file comprising layering violation metrics for the architecture of a software system.

6 Results Analyzes

This section reports and discusses the measurement results for the architecture layer violation principles. Table 2 shows, for each layer in both OO and AO version along the selected releases, the number of skip-call violations (SC), back-call violations (BC), number of modules with skip-call violations (NMSC), number of modules with back-call violations (NMBC), total number of modules. Table 3 also contains the back-call violation index (BCVI) for the entire system: BCVI(S), as well as the skip-call violation index (SCVI) for the entire system: SCVI(S). It is desirable that these

two indexes have the value 1 (maximum value), meaning that there are neither back-calls nor skip-calls in the entire system. A value near to 0 indicates violation to large extent. More information about these metrics can be found elsewhere [17]. In this study, we did not measure cyclical dependence violations, because the cycles identified in the selected releases we analyzed were between modules of the same layer, which is not a violation.

Table 2. Evaluation of architectural violations present in HW scenarios.

		SC		**BC**		**NMSC**		**NMBC**		**TNM**	
		OO	AO	OO	AO	OO	AO	OO	AO	OO	AO
Release 1	GUI	227	171	0	0	18	19	0	0	25	25
	Communication	10	10	0	0	2	2	0	0	5	7
	Business	26	15	12	0	5	3	1	0	9	8
	Data	0	0	0	0	0	0	0	0	28	28
	Reusable Elem.	0	0	0	0	0	0	0	0	21	21
Release 2	GUI	227	171	0	0	20	21	0	0	29	29
	Communication	10	10	0	0	2	2	0	0	5	7
	Business	26	15	12	0	5	3	1	0	9	8
	Data	0	0	0	0	0	0	0	0	28	28
	Reusable Elem.	0	0	0	0	0	0	0	0	21	25
Release 3	GUI	227	171	0	0	20	21	0	0	29	29
	Communication	10	10	0	0	2	2	0	0	5	7
	Business	26	15	12	0	5	3	1	0	9	8
	Data	0	0	0	0	0	0	0	0	28	28
	Reusable Elem.	0	0	0	0	0	0	0	0	33	38
Release 4	GUI	233	171	0	0	20	21	0	0	29	29
	Communication	10	10	0	0	2	2	0	0	5	7
	Business	26	15	15	0	5	3	1	0	9	8
	Data	0	0	0	0	0	0	0	0	28	28
	Reusable Elem.	0	0	24	0	0	0	3	0	35	37
Release 5	GUI	233	171	0	0	20	21	0	0	29	29
	Communication	10	10	0	0	2	2	0	0	8	7
	Business	26	15	0	0	4	3	0	0	8	8
	Data	0	0	0	0	0	0	0	0	28	28
	Reusable Elem.	0	0	24	0	0	0	3	0	35	37

* SC: Skip-Call violation; BC: Back-Call violation; NMSC: Number of Modules with Skip-Call violation; NMBC: Number of Modules with Back-Call violation; TNM: Total Number of Modules.

An interesting point to notice is that, in all releases, the number of skip-call in AO version is smaller. In fact, many skip-calls in the GUI layer are due to the handling of exceptions signaled by data layer, such as `ObjectAlreadyInsertedException`. Besides, there are also some skip-calls to the reusable elements layer, such as basic classes. In fact, the GUI layer provides text field components so that the user can register a complaint. The Servlets obtain the values from the text fields and instantiates a `Complain` object, resulting in a skip-call from GUI to reusable elements layer. In AO version, some exceptions thrown in data layer are treated with aspects, which uses AspectJ `declare soft` construct so that is not necessary to treat some exceptions in GUI layer. However, the exception is treated within the appropriate aspects, which does not result in skip-call violation because aspects do not belong to

any layer (as explained in Section 4). This contributes to the smaller occurrence of skip-calls in AO than in OO version.

Table 3. Skip-Call and Back-Call Violation Indexes in HW releases.

	Release 1		**Release 2**		**Release 3**		**Release 4**		**Release 5**	
	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO
BCVI(S)	0.972	1	0.972	1	0.972	1	0.950	1	0.978	1
SCVI(S)	0.575	0.528	0.626	0.581	0.626	0.581	0.624	0.580	0.766	0.58

* BCVI(S): Back-Call Violation Index; SCVI(S): Skip-Call Violation Index.

The analysis made by this work found an interesting skip-call, that revealed to be business code mixed with GUI code. This is a very trivial issue, which occurred in a single location, and could be easily fixed by any developer. In this case, a business validation is done in a Servlet (GUI layer), whereas it should be done in a business layer module. This example shows that using the framework also helps to identify possible mistakes that can violate layer principles and might be passed unnoticed otherwise.

In business layer, there are skip-calls to reusable elements layer, because some record classes, such as `ComplaintRecord`, need to call methods from basic classes to verify if the object is already inserted in the `ComplaintRepository` class (data layer). In AO version, the number of skip-calls in business layer is smaller. In this case, there are some skip-calls to `ConcurrencyManager` module from business layer in OO, but in AO an aspect responsible for synchronization (`HWManaged-Synchronization`) takes care of such calls.

Even though the number of skip-calls in AO is smaller than in OO version, the SCVI(S) metric indicates the skip-call violations in AO version is more problematic than in OO. The reason is because SCVI(S) is based on the proportion of the number of skip-call compared to the total number of calls for each layer.

Some interesting back-calls are presented in OO version 4. In this case, the observer pattern [17] requires that basic classes (reusable elements layer) add a notify method, which treats repository exceptions (data layer). In AO version, on the other hand, this pattern is implemented with aspects, and therefore there are no back-calls. Many of these violations could be solved by a simple refactoring. For instance, the skip-calls from GUI modules that treat repository exceptions could be avoided by creating business exceptions that can be thrown whenever a repository exception appears. This new exception can contain business information and the repository exception can be passed as its cause. In fact, all violations can be easily checked by using our framework. This is a useful tool during software development for quickly identifying layer violations principles that could be passed unnoticed by the developer otherwise.

7 Conclusions and Future Works

In this work, we analyzed the differences between AOP and OOP in the Health Watcher System. In both versions, we could find some architecture layer violations that could suggest refactoring in order to conform to the layer architecture. We analyzed how AOP mechanisms influence the metrics and we motivate suggestions to

extend these metrics to analyze the dependency of aspects in layer architecture, quantitatively.

We discussed the problems involving the documentation related to the placement of certain modules into layers. In fact, this task is very important, since it directly influences the metrics on violation principles. Sometimes this task is not so simple, especially when dealing with reusable elements and aspects.

This study has some important limitations. However, this is an ongoing work and we expect to use the feedback from the workshop to improve the following restrictions. First, we are not taking into account the aspects dependency, since aspects do not belong to any layer. Second, besides the five selected evolution scenarios analyzed in this study, we can also evaluate the others evolution scenarios provided in [8], in order to perform conclusions more consistent and representative. Finally, we also pretend to apply these metrics in different systems in order to assess with more evidences on the impacts of AOP in layer architecture violation principles.

References

1. Bass, L., P. et al. Software Architecture in Practice, 2nd edition, Addison-Wesley, 2003.
2. Buschmann, F., et al. Pattern-Oriented Software Architecture: A System of Patterns. Wiley, 1996
3. Cheesman, J., et al: A Simple Process for Specifying Component-Based Software. Addison-Wesley, October 2000.
4. Cormen, T. H. et al. Introduction to Algorithms. 2nd Edition, MIT Press, 2001.
5. Dijkstra, E. W. The structure of THE-multiprogramming system. Communications of ACM, 11(5):341-346, 1968.
6. Figueiredo, E. et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. Proceedings of ICSE'08, Leipzig, Germany, 2008.
7. Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, 1995.
8. Greenwood, P. et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. Proc. of Ecoop.07, Berlin, Germany, 2007.
9. Kiczales, G. et al. Aspect-Oriented Programming. Proceedings of ECOOP'07, LNCS 1241, Springer, pp. 220-242, 1997.
10. Kiczales, G., et al. Getting Started with AspectJ. Communications of the ACM, 44(10):59–65, October 2001.
11. Kulesza, U. et al, Quantifying the Effects of AOP: a Maintenance Study. Proceedings of ICSM'06, Philadelphia, Sep 2006.
12. Lobato, C., et al. Evolving and Composing Frameworks with Aspects: The MobiGrid Case. In Proceedings of ICCBSS 2008 - Volume 00 IEEE Computer Society, 53-62, 2008.
13. McClain, G. R., ed., Open Systems Interconnection Handbook. New York, NY: Intertext Publications McGraw-Hill Book Company, 1991.
14. Mezini, M. and Ostermann, K. Conquering Aspects with Caesar. Proc. of AOSD, pp. 90-99, Boston, USA, 2003.
15. Molesini, A., et al. On the Quantitative Analysis of Architecture Stability in Aspectual Decompositions. In Proceedings of WICSA 2008 - IEEE Computer Society, 29-38.
16. Rumbaugh, J., Jacobson, I., and Booch, G. 2004 Unified Modeling Language Reference Manual, the (2nd Edition). Pearson Higher Education.
17. Sarkar, S., et al. A Method for Detecting and Measuring Architectural Layering Violations in Source Code. In Proceedings of APSEC. IEEE Computer Society, DC, 165-172.
18. Soares, S., et al: Implementing distribution and persistence aspects with AspectJ. In Proceedings of. OOPSLA '02. ACM, New York, NY, 174-190.

LTS-based Semantics and Property Analysis of Distributed Aspects and Invasive Patterns*

Luis Daniel Benavides Navarro, Rémi Douence, Angel Núñez, Mario Südholt

OBASCO project; EMN-INRIA, LINA
Dépt. Informatique, École des Mines de Nantes
4 rue Alfred Kastler, 44307 Nantes cédex 3, France
{lbenavid,douence,anunez,sudholt}@emn.fr

Abstract. Invasive patterns are an extension of standard parallel and distributed architectural patterns for complex distributed algorithms. They have previously been implemented in terms of the AWED language, an aspect language with features for explicit distribution. In this paper we present two formal semantics based on labeled transition systems, one for AWED and the other for invasive patterns, for the definition of interaction properties of aspects and pattern compositions. We also show how the semantics can be used to check corresponding liveness and safety properties.

1 Introduction

In previous research we investigated how invasive patterns [1] that are defined on top of AOP languages can be used to address the complexity of distributed algorithms in real industrial middleware for replicated caching. We have implemented invasive patterns by means of a formal transformation into AWED [2], an aspect language with features for explicit distribution. The transformation defines invasive patterns in terms of interacting AWED aspects. This paper introduces two direct formal semantics, one for AWED and the other for invasive patterns. Furthermore, we harness these semantics to formally show, for the case of AWED, how remotely deployed aspects interact, and how individual pattern definitions interact in the case of invasive patterns.

During the design of invasive patterns we argued that distributed algorithms, and specific properties over those algorithms, can be analyzed and enforced easily in terms of invasive pattern abstractions. For example, Fig. 1 presents a high-level pattern-based view of the system structure of JBoss Cache's replication algorithm with transactions with pessimistic locking and a two phase commit protocol. A transaction is triggered by a specific method call represented by the first node in the pattern. Then successive calls to `get`, `remove` or `put` methods on the cache are executed and the information is stored for further replication. When a particular value is not present in the cache, the latter looks for the value

* Work partially supported by AOSD-Europe, the European Network of Excellence in AOSD (www.aosd-europe.net).

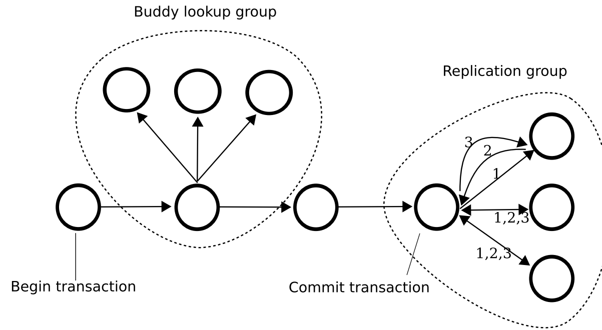

Fig. 1. Architecture of transaction handling with replication in JBoss Cache

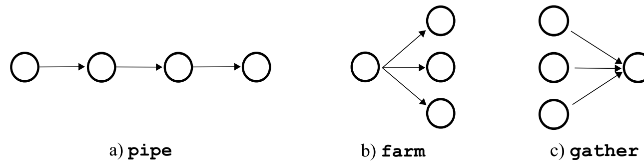

Fig. 2. Basic patterns

in a group of selected neighboring nodes, its so-called buddies, illustrated by the three edges starting in the second node of the figure. Once the end of a transaction is reached, the originating cache engages a two phase commit protocol. As part of this protocol, the originating cache sends a prepare message with the transaction control information (edges numbered 1), followed by answers from all buddies confirming agreement or non agreement (edges numbered 2). Finally, the originating cache sends a final commit or a rollback message depending on the answers it received (edges numbered 3).

As we have shown, the two phase commit protocol implementation can naturally be expressed in terms of compositions of invasive patterns, that is, extensions of the three basic patterns showed in Fig. 2 that account for highly crosscutting references of the patterns to contextual information. The steps denoted 1–3 in Fig. 1 correspond, for instance, to two applications of the `farm` pattern and one application of the `gather` pattern. Analyzing properties over such pattern specifications is more feasible than analyzing properties over current Java implementation of JBoss Cache (more than 50000, highly entangled, lines of code) where this architecture is essentially hidden. For that purpose, this paper presents three contributions: a formalization of AWED, a formalization of invasive patterns, both based on Labeled Transitions Systems (LTS), and some examples of how to use the LTS-based semantics with LTSA [3] (http://www.doc.ic.ac.uk/ltsa/eclipse) as analyzing tool.

2 The AWED Language

AWED supports AOP in a distributed context. A pointcut can match joinpoints on different hosts. A sequence of joinpoints can involve different hosts. An advice can be executed remotely, synchronously or asynchronously to the base execution. Furthermore, pointcuts can include predicates over groups of hosts.

The grammar shown in Fig. 3 shows the essentials of pointcut definitions in the AWED language (the full language definition can be found in [4]). The pointcut language allows matching of method calls (terminal `call`), nested calls (`cflow` means control-flow) and arbitrary (regular) sequences of method calls (non-terminal *Seq*). The constructors `host` and `on` specify (groups of) hosts where a pointcut is matched (or where an advice is executed). The constructors `target` and `args` bind values (such as the receiver or the arguments of a method call) to variables. This enables values to be passed from a matching execution event to the corresponding advice. Pointcuts can be composed using logical operators (union, intersection and complement). Sequences (*Seq*) are defined in terms of transitions of non-deterministic finite-state automata. An automaton is a set of transitions *Step*. Each transition has a label *id* and its pointcut *Pc* non-deterministically leads to a set of *Id*. The constructor `step` identifies the transition in the automaton that should trigger the advices.

```
// Pointcuts
Pc     ::= call(MSig) | cflow(Pc) | Seq
         | host(Group) | on({ Hosts })
         | target({Type}) | args({Arg})
         | Pc || Pc | Pc && Pc | !Pc
Seq    ::= Id: seq({Step}) | step(Id,Id)
Step   ::= Id: Pc → Target
Target ::= Id || ... || Id
Hosts  ::= localhost | jphost | "Ip:Port" | GroupId
```

Fig. 3. The AWED language (excerpts)

3 Pattern Language

Figure 4 shows the syntax of invasive patterns language (we have omitted details for the sake of simplicity).

The pattern constructor `patternSeq` takes as argument a list G_1 A_1 G_2 A_2 $\ldots$ G_n of alternating group and aspect definitions. Each triple G_i A_i G_{i+1} in this list corresponds to a pattern application that uses the aspect A_i to trigger the pattern in a source group G_i and realizes effects in the set of target hosts G_{i+1}. A group G is either defined as a set of host identifiers H or through a pattern constructor term itself. In the latter case, the group is defined as the source or target group of the constructor term depending on the argument position the term is used in. This constructor enables the definition of the basic patterns shown in Fig. 2. Pattern compositions can be defined with more complex `patternSeq` terms. For instance, the left hand side of Fig. 5 defines , using a

P ::= `patternSeq` $G_1\ A_1\ G_2\ A_2 \ldots G_n$
G ::= $H\ G$ | $P\ G$ | ϵ
A ::= `aspect` { `around`((H, Id*)*): PCD *SourceAdvice* [`sync`] *TargetAdvice* }
PCD ::= `call`(*MSig*) | `target`(*Id*) | `args`(*Id*+)
| PCD && PCD | PCD || PCD | !PCD
| *Seq*

Fig. 4. Pattern language

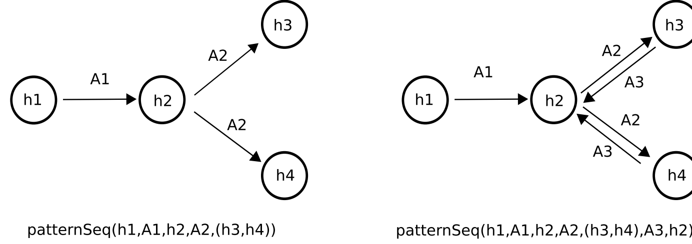

Fig. 5. Pattern Compositions

concrete syntax, a composition `pipe` then `farm`, and its right hand side defines a composition `pipe`, `farm` then `gather`.

Aspects A specify a pointcut PCD that allows the modularization of the crosscutting code that triggers a pattern, and define a source advice and a target advice executed respectively on the source and target groups of a pattern. Advice can be parametrized by source hosts H and bound values (see `args` below). A source advice can call the matched base call with the `proceed` keyword. Otherwise, the base call triggers the aspect but the execution of the corresponding base method is skipped. When a `sync` annotation is used to qualify target advice, the base program execution on source hosts is not resumed before the end of the target advice. The default behavior is asynchronous execution.

Pointcut definitions that, for presentation purposes, are essentially restricted to matching of method call joinpoints, may extract target objects with `target` and arguments of calls with `args` and use logical compositions of pointcuts. Following the paradigm of stateful pointcuts [5, 2] (and unlike AspectJ [6, 7]), pointcuts may match sequences (non-terminal *Seq*) of calls in the base program execution. We omit the syntax of sequences for presentation purposes.

4 Formal Semantics for Language Constructs

This section formally defines the semantics of our language constructs in terms of labeled transition systems (LTSs). It also shows how these definitions can be analyzed with model checkers such as LTSA. More precisely, we use Finite State Processes (FSP) [3], a textual description of LTSs. In the remainder we use the terms FSP and LTS interchangeably. We first focus on AWED in Sec. 4.1, then we extends its formalization for invasive patterns in Sec. 4.2.

```
1  // local aspect (and advice) to de deployed on h_1
2  Local = (h_1.loadInCache -> h_1.proceed ->
3          h_2.h_1.remoteAdvGo -> h_3.h_1.remoteAdvGo -> Local).
4
5  // remote advice to be deployed on h_2
6  RemoteAdv1 = (h_2.h_1.remoteAdvGo -> h_2.loadInCache -> RemoteAdv1).
7
8  // remote advice to be deployed on h_3
9  RemoteAdv2 = (h_3.h_1.remoteAdvGo -> h_3.loadInCache -> RemoteAdv2).
10
11 ||Aspect = (Local || RemoteAdv1 || RemoteAdv2).
```

Fig. 6. Semantics of a Cache Replication AWED Aspect

4.1 Formalization of (AWED) distributed aspects

AWED provides aspects in a distributed context by introducing two notions: group of hosts, and asynchronous remote advice. In AWED, an aspect specifies a group of hosts for pointcut and another group of hosts for remote advice. When a joinpoint occurs on *any* host of the pointcut group, the local advice is executed on this host, and the remote advice is executed on *each* host of the remote advice. Moreover, the remote advice is executed asynchronously (by default) and the local base program resumes its execution as soon as the remote advices are started. For instance, consider the following example:

```
aspect execution(void {h_1}.loadInCache(URL)) && args(url)
      { proceed(url); }
      { {h_2,h_3}.loadInCache(url)}
```

This distributed AWED aspect replicates the update on hosts `h_2` and `h_3` cache updates of host `h_1`. Indeed, when the pointcut corresponding to `loadInCache` is matched on `h_1`, the local advice `proceed(url)` performs the local cache update and the remote advice updates the caches on hosts `h_2` and `h_3`. Such an AWED aspect can be formally defined in LTS as in Fig. 6. An aspect is defined as the parallel composition of one LTS by host. An LTS is a finite state automaton with labeled transitions. For instance, in Fig. 6 there are three LTSs, the first one `Local` defines a sequence of four transitions each separated by `->`. The `Local` LTS models the aspect: when the joinpoint occurs on the first host (*i.e.*, `h_1.loadInCache`), the method is executed locally (*i.e.*, `h_1.proceed`), then the remote advices are started on the second and third hosts (*i.e.*, `h_2.h_1.remoteAdvGo` and `h_3.h_1.remoteAdvGo`). Then this recursive definition waits for the next occurrence of the pointcut. The definition `RemoteAdvice1` models the remote advice on the second host: it waits for `remoteAdvGo` to execute the replicated cache loading and waits again. The remote advice `RemoteAdvice2` on the third host is similar. The semantics of the full aspect is defined by the parallel composition (noted `||`) of these three LTSs. The resulting LTS `Aspect` is the synchronized product of the three automata and it implements rendez-vous for shared transitions.

The semantics of the woven program is defined by the parallel composition of the base program specification with the aspect:

```
Base       = (h_1.loadInCache -> Base).
||Woven    = (Base || Aspect).
```

Such a formalization enables us to check for properties with LTSA. For instance, the following property `P` states that every cache loading on the first host is executed locally, then replicated on the second host.

```
property P = (h_1.loadInCache -> h_1.proceed -> h_2.loadInCache -> P).
||Check_P  = (Woven || P).
```

In fact, the property `P` is violated by our system that allows the execution trace:

```
h_1.loadInCache
h_1.proceed
h_2.h_1.remoteAdvGo
h_3.h_1.remoteAdvGo
h_1.loadInCache
```

Indeed, the second occurrence of `loadInCache` can happen before the first one has been replicated on the second host because remote advices are asynchronous. We can make synchronous the remote advices by adding an event `remoteAdvEnd` at the end of each remote advice, and by waiting for these events in the local aspect:

```
Local = (h_1.loadInCache -> h_1.proceed ->
          h_2.h_1.remoteAdvGo -> h_3.h_1.remoteAdvGo ->
            h_2.h_1.remoteAdvEnd -> h_3.h_1.remoteAdvEnd -> Local).
```

In this case, the property `P` is satisfied.

The generic translation from the AWED aspect to LTS is defined in Fig. 7. We use a `for all` notation in order to enumerate hosts in each group. The translation generalizes the previous example by introducing indexes (*e.g.*, at line 6, `Local_i_j` is indexed by `i` from `G_i`) in order to avoid spurious synchronization.

4.2 Formalization of Invasive Patterns

Invasive patterns introduce two more notions. First, an invasive pattern defines a sequence of aspects. Initially, only the first aspect of the sequence is active. Once this first aspect has crosscut, and its remote advice is executed, then the second aspect becomes active. And so on. This is formalized by adding a waiting loop at the beginning of each aspect in order to ignore the joinpoint when the aspect is not active. Moreover, each aspect in a sequence of aspects, waits for an event `step_`i to become active and its remote advice generates an event `step_`$i+1$ to activate the next aspect, where i is the rank of the aspect in the sequence. Second, quite similarly to AWED, each aspect in an invasive pattern relates two groups of hosts `G1` and `G2`, but each aspect in an invasive pattern requires a rendez-vous in order to execute a remote advice. Indeed, a remote advice of an

```
1  // semantics of
2  // G_i.evt LADV G_{i+1}.RADV
3
4  // a full aspect i
5  ||Aspect_i = (
6    (for all h_j in G_i. ||Local_i_j)
7    ||
8    (for all h_k in G_{i+1}.
9      || RemoteAdvice_i_k)
10 ).
11
12 // local aspect (and advice)
13 // to de deployed on h_j in G_i
14 Local_i_j = (
15
16
17
18
19
20   // pointcut
21   h_j.evt ->
22   // local advice
23   h_j.LADV_i ->
24   // start remote advices
25   for all h_k in G_{i+1}.
26     h_k.h_j.remoteAdvGo_i -> ...
27   // wait for end of remote advices
28   // only if synch remote advice
29   for all h_k in G_{i+1}.
30     h_k.h_j.remoteAdvEnd_i -> ...
31   // and start again
32   Local_i_j
33
34 ).
35
36 // remote advice to be deployed
37 // on h_k in G_{i+1}
38 RemoteAdvice_i_k = (
39   // wait for start by any h_j
40   for all h_j in G_i.
41     | (h_k.h_j.remoteAdvGo_i ->
42        // remote advice
43        h_k.RADV_i ->
44        // end of synch remote advice
45
46        h_k.h_j.remoteAdvEnd_i ->
47
48
49        // and start again
50        RemoteAdvice_i_k)
51 ).
```

Fig. 7. Semantics of AWED Aspects

```
// semantics of
// G_i evt LADV RADV G_{i+1}

// a full aspect
||Aspect_i = (
  (for all h_j in G_i.||Local_i_j)
  ||
  (for all h_k in G_{i+1}.
    || RemoteAdvice_i_k)
).

// local aspect (and advice)
// to de deployed on h_j in G_i
Local_i_j = (
* // waiting loop
* h_j.evt -> Local_i_j
* |
* // my turn
* h_j.step_i -> (
    // pointcut
    h_j.evt ->
    // local advice
    h_j.LADV_i ->
    // start remote advices
    for all h_k in G_{i+1}.
      h_k.h_j.remoteAdvGo_i -> ...
    // wait for end of remote adv.
    // only if synch remote advice
    for all h_k in G_{i+1}.
      h_k.h_j.remoteAdvEnd_i -> ...
    // and start again
    Local_i_j
* )
).

// remote advice to be deployed
//on h_k in G_{i+1}
RemoteAdvice_i_k = (
* // rendez vous
* for all h_j in G_i.
*   h_k.h_j.remoteAdvGo_i -> ...
  // remote advice
  h_k.RADV_i ->
  // end of synch remote advice
* for all h_j in G_i.
    h_k.h_j.remoteAdvEnd_i -> ...
* // activate the next aspect
* h_k.step_{i+1} ->
  // and start again
  RemoteAdvice_i_k
).
```

Fig. 8. Semantics of invasive patterns

invasive patterns is executed on a host of `G2` only when a pointcut has been detected on *each* host of `G1`.

The formalization of an invasive pattern from the group `G_i` to the group `G_i+1` is defined in Fig. 8. It is an extension of the AWED semantics in Fig. 7: the lines marked with a star `*` are new lines. First, each aspect starts by a waiting loop in order to ignore joinpoint when the aspect is not active (lines 15 and 16). Line 17 specifies a choice noted `|` with a second branch: an aspect becomes active, when the right `step` of the sequence has been reached (lines 18 and 19). A remote advice starts now by a rendez-vous of all hosts of the group `G_i` (lines 39-41). The next step of the sequence is activated (with respect to the current remote host `h_k`) only when the remote advice is over (lines 47 and 48).

This formalization can be used to (model) check properties as in the previous section. For example, consider again a replicated cache where the following invasive pattern implements a distributed two phases commit protocol with three aspects:

```
G_a prepare { proceed } { prepareR } ;
G_b true    { }         { }          ;
G_a true    { commit }  { commitR }  G_b
where
G_a = {h_1}, G_b = {h_2,h_3}
```

The two phases of the protocol are: `prepare` then `commit`. Both phases must be replicated on remote hosts. The first aspect replicates the first phase: it crosscuts `prepare` on `G_a` (*i.e.*, `h_1`), performs locally this function and its remote advice performs `prepareR` on `G_b` (*i.e.*, `h_2` and `h_3`) and activates the second aspect. The second aspect synchronizes the end of the first phase and the beginning of the second phase on different groups (`G_b` and `G_a`): it does not wait for a particular joinpoint (*i.e.*, its pointcut is `true`) and it does not perform a particular action (its advices are empty) but as soon as it has been activated it activates the third aspect on `G_a`. Finally, the third aspect performs the second phase: it does not wait for a particular joinpoint (*i.e.*, its pointcut is `true`) but as soon as it has been activated it performs `commit` locally on `G_a` and `commitR` remotely on `G_b`.

The formalization of this example in LTS can be checked with LTSA. For instance, the following property verifies that the two phases on `h_1` are replicated on `h_2`:

```
P2 = (h_1.prepare-> h_2.prepareR_1-> h_1.commit_3-> h_2.commitR_3-> P2).
```

In fact the semantics does not satisfy this property. Indeed, there is a waiting loop at the start of the first aspect that can ignore an arbitrary number of joinpoints. For instance, the short trace

```
h_1.prepare
h_1.step
```

leads to a progress violation, since the first aspect is initially inactive and it can ignore the first occurrence of `prepare`. So, in this case this first event is not replicated (it does not trigger `h_2.prepareR_1`). In order to take this into account, the property must be slightly modified as follows:

```
P2 = ( h_1.prepare -> P2
     | h_2.prepareR_1 -> h_1.commit_3 -> h_2.commitR_3 -> P2).
```

An alternative option would be to remove the waiting loop at the start of the first aspect in order to take into account all occurrences of the first phase.

Our semantics can also be used to check invasive pattern definition equivalence. For instance, the previous invasive pattern seems equivalent to

```
G_a prepare { proceed  } { }         ;
G_b true    { prepareR } { }         ;
G_a true    { commit }   { commitR } G_b
```

The remote advice `prepareR` of the first aspect can be slightly delayed by shifting its code to the local advice of the second aspect. Indeed, the pointcut of the second aspect is `true`. So its local advice is executed as soon as the aspect is activated. However this is not equivalent: the second aspect is activated as soon as all remote advices of the first aspect are over (*i.e.*, there is a rendez-vous at the end of remote advices). So, our original definition enforces that all `prepareR` are over before the protocol proceed, while this alternative definition does not. We discovered such a difference while checking for other properties with LTSA.

5 Related Work

Event-based AOP [5, 8] is a general framework for AOP where pointcuts relate sequences of events. The framework is defined formally, allowing to automatically deduce possible malicious interactions between aspects. The interaction analysis remains applicable to Invasive Patterns that could be considered as a subset of EAOP. Tracematches [9] extends AspectJ with aspect-like constructs to detect sequences of joinpoints with free variables. The interaction of several tracematches is determined by the *precedence* mechanism of AspectJ. Our formalism allows a fine grained control over aspect precedence and pattern composition, providing also a model to detect sequence of joinpoints in distributed settings. Concurrent Event-based AOP (CEAOP) [10] uses LTS(A) to model base program, concurrent aspects, and their interactions. Our use of LTS(A) is analogous, but with a special focus on distribution. Caromel *et. al.* introduce a concurrent object calculus [11] implemented by the Java library ProActive [12]. Different from LTS, asynchronous communication and *futures* is at the root of this calculus. Katz's work [13] and Djoko's work [14] classify aspects in terms of the safety and liveness properties of the base program that are preserved after weaving a single type of aspect. We contribute in the verification of safety and liveness properties that are the result of complex interaction between several aspects in a distributed setting.

6 Conclusion

In previous publications we have presented AWED and invasive patterns. This paper specifies their semantics using labeled transition systems. It clarifies the

different interactions that these aspect languages introduce. Thus, it allows the verification of safety and liveness properties of the woven program, such as *deadlock freedom* and *progress*, using the LTSA tool. This paper is a step towards a complete specification that should consider specific aspect-based concurrency constructs. In this regards, an integration with CEAOP is seen as future work.

References

1. Benavides Navarro, L.D., Südholt, M., Douence, R., Menaud, J.M.: Invasive patterns for distributed programs. In: Proc. of the 9th Int. Symp. on Dist. Obj., Midd., and App. (DOA'07), Springer Verlag (November 2007) 772–789
2. Benavides Navarro, L.D., Südholt, M., Vanderperren, W., Fraine, B.D., Suvée, D.: Explicitly distributed aop using awed. In: AOSD '06: Proc. of the 5th international Conf. on Aspect-oriented software development, New York, NY, USA, ACM Press (2006) 51–62
3. Magee, J., Kramer, J.: Concurrency: State Models and Java. 2nd edn. Wiley (2006)
4. Benavides Navarro, L.D., Südholt, M., Vanderperren, W., Verheecke, B.: Modularization of distributed web services using awed. In: Proc. of the 8th Int. Conf. on Distributed Objects and Applications. LNCS 4276, Springer Verlag (October 2006) 1449–1466
5. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: Proc. of GPCE'02. Volume 2487 of LLNCS., Springer-Verlag (October 2002) 173–188
6. AspectJ: Aspectj home page. http://www.eclipse.org/aspectj (2008)
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: ECOOP '01: Proceedings of the 15th European Conf. on Object-Oriented Programming. Springer-Verlag, London, UK (2001) 327–353
8. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: Aspect-Oriented Software Development (AOSD04), ACM, ACM Press (March 2004) 141–150
9. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. SIGPLAN Not. **40**(10) (2005) 345–364
10. Douence, R., Botlan, D.L., Noyé, J., Südholt, M.: Concurrent aspects. In: GPCE '06: Proc. of the 5th Int. Conf. on Generative programming and component engineering, New York, NY, USA, ACM (2006) 79–88
11. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. SIGPLAN Not. **39**(1) (2004) 123–134
12. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, deploying, composing, for the grid. In Cunha, J.C., Rana, O.F., eds.: Grid Computing: Software Environments and Tools. Springer-Verlag (January 2006)
13. Katz, S.: Aspect categories and classes of temporal properties. Trans. on Aspect Oriented Software Development (TAOSD) (2006) 106–134 LNCS 3880.
14. Djoko, S.D., Douence, R., Fradet, P.: Aspects preserving properties. In: ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'08), San Francisco, CA, USA, ACM Press (January 2008) 135–145

Using Transformation-Aspects in Model-Driven Software Product Lines*

Hugo Arboleda[1,2], Rubby Casallas[2], and Jean-Claude Royer[1]

[1] École des Mines de Nantes, 4 rue Alfred Kastler, 44307 Nantes Cedex 3, France
{hugo.arboleda,jean-claude.royer}@emn.fr
[2] University of Los Andes, Cra. 1 N◦ 18A 10, Bogotá, Colombia
rcasalla@uniandes.edu.co

Abstract. Model-Driven Software Product Lines (MD-SPL) are configured by using configuration models and Problem Space metamodels that capture product line scope. Products are derived by means of successive model transformations, starting from problem space models and based on the configuration models. Fine-variations of MD-SPLs correspond to characteristics that affect particular elements of models involved in the model transformations. In this paper, we present an approach to create MD-SPL including fine-variations. We configure products creating fine-feature configurations. Then, based on such configurations, we create MD-SPLs using principles of Aspects Oriented Development. Thus, our approach allows to derive products including fine-grained details of configuration.

Key words: Model Driven Development, Software Product Lines, Variability Management, Product Derivation, Fine-Grained Variability.

1 Introduction

Aspect-oriented programing (AOP) [1–3] improves software development by providing constructs for the encapsulation of crosscutting concerns. Aspects encapsulate crosscutting concerns and are subsequently composed with other software artifacts using composition mechanisms. A join point model captures the set of possible composition points for a specific aspect. Aspects are automatically composed with the rest of the system by an aspect weaver. Asymmetric AOP approaches such as AspectJ [1] provide constructs for the encapsulation of crosscutting concerns that are woven to some (non-AOP) base system. A recent work [4, 5] has shown that AOP is a valuable complement for Model Driven Development (MDD) [6] regarding the automatic generation of software products. First, model transformation programs can be modularly adapted [5, 7], based on approaches as the one introduced by AspectJ. Second, Aspect Oriented Modeling (AOM) techniques [8–11] provide support to implement variability at model

* This work has been supported by the European Commission STREP Project AMPLE IST-033710 and the "Instituto Colombiano para el Desarrollo de la Ciencia y la Tecnología - Francisco José de Caldas (COLCIENCIAS)"

level. This means that different models can be adapted applying a weaving that crosscuts their concerns.

MDD has proved benefits regarding the derivation of Software Product Lines (SPLs). In Model Driven Software Product Line approaches (MD-SPL) [12–14], product families are derived by means of chains of model transformations, starting from an initial domain model and based on configuration models, until to get a final solution model. Current Aspect Oriented (AO-)MD-SPL approaches, as those introduced by Voelter et al. [7], deal with the problem of creating products including variations that affect the whole product. We call these, *coarse-grained variations*. For example, a coarse-grained variation is a property as internationalization (English or German), or in a SPL of Smart Home systems, a coarse-grained variation expresses that a house has automatic lights. This implies that for a particular Smart Home system, "all" the instances of a light component have functionality of automatic lights.

In our work, it is important to distinguish *coarse-grained variations* from *fine-grained variations*. Fine-grained variations are characteristics that affect particular elements of models involved in model transformations. For example, for a particular Smart Home system, it is possible to select the instances of a light component that have functionality of automatic lights. In addition, it is possible that such instances manage different attributes to configure the intensity of the lights when they are turned on. Thus, it is possible to have an automatic light in the living room that is turned on at 19h00, and another automatic light in the main room that is turned on in the presence of an inhabitant. The expression of fine-grained variations and the use of it to subsequently derive products are important activities in the SPLs creation process. This is because customer requirements can be very specific, especially in the domain of software intensive systems where, *(i)* each component instance can manage variable functionality, and *(ii)* variability related to each component instance can imply expenses for customers.

In this paper, we improve the AO-MD-SPL approach presented by Voelter et al. [7], facilitating the creation of products including fine-grained variations. We express fine-grained variations and define its scope by using *constraint models*; and, we configure products including fine-grained variations by using *fine-feature configurations*. We also include a solution for the problem of deriving products based on fine-grained details of configuration. For this, we adapt the execution of model transformations applying *aspects* that include transformation logic to generate low-level configuration details. The aspects we create use a particular type of transformation rules that we have called *fine-transformation rules*. For instance, we adapt the execution of a model transformation that transforms elements conforming to a metaconcept `Room`, when some of such elements is affected by a fine-grained variation. The fine-grained variation can be that a `Room` element has to be transformed including an environmental control system as air conditioning.

We have implemented our approach using and extending the openArchitectureWare (oAW) [15] toolkit. We have chosen oAW, because this is a complete

MDD framework integrated into Eclipse that makes possible the reading and instantiating of models, checking them for constraint violations, and transforming them into other models or source code. Furthermore, oAW also provides support for AOM and AOSD in the context of MDD. Finally, the framework has been used successfully to create SPLs [5], and there is an active community of developers using and improving it [15].

The remainder of this paper is organized as follows. Section 2 introduces a practical example we use to illustrate our approach. Section 3 presents our proposal for expressing fine-grained variations and configure products including it. Section 4 explains our solution to create MD-SPLs including fine-grained variations. Section 5 presents conclusions.

2 Practical Example

In the remaining of this paper, we will use as example the case of a Smart Home Product Line which is part of the AMPLE Project [16]. The description of a house includes structural elements as floors, rooms, doors, windows and stairs. Houses also include electrical and electronic devices such as automatic lights, security devices as alarms, among others. These devices, and therefore their behavior, are optional features that product designers have to select and bind to other elements that already exist in the house. For instance, to derive a particular Smart Home system, the automatic lights can be bound to all rooms in the house (coarse-grained variation) and the security alarm system can be bound only to the main door entrance (fine-grained variation). Figure 1 presents part of a domain model of a specific Smart Home system used as example.

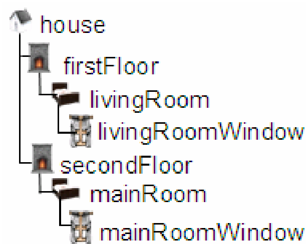

Fig. 1. Domain Model of a Smart Home.

The model is represented by a tree structure created using EMF [17]. This structure illustrates UML-type composition, for example, between the `house` and its `floors`. In the figure there are two floors, `firstFloor` and `secondFloor`. The `firstFloor` has a `livingRoom` and a `livingRoomWindow`; the `secondFloor` has a `mainRoom` and a `mainRoomWindow`. This model conforms to a domain metamodel that includes concepts of smart homes like <<Building>>, <<Floor>>, <<Room>> and <<Window>>.

3 Fine-Grained Variability

We distinguish between coarse-grained variations and fine-grained variations. The sentence *all the doors have fingerprint authentication system*, is an example of coarse-grained variation, and one of fine-grained variation is, *only the*

main door has a fingerprint authentication system. To express and manage fine-variations, we introduce *constraint models*. To configure products including fine-variations, we introduce *fine-feature configurations*. We based the creation of constraint models and fine-feature configurations on feature modeling. In addition, we use feature modeling to express coarse-grained variations.

3.1 Feature Modeling

The basic mechanism we use to classify and describe configurable common and variable characteristics in the scope of a product line is feature models [18]. Specifically, we use the Czarnecki et al. [13] feature metamodel. In this feature metamodel, a `GroupFeature` expresses a choice over the set of `GroupedFeature`s in the group and its `cardinality` defines the restriction on the number of choices. For example, the `cardinality` [1..2][4..5] for a `GroupFeature` means that between one and two, and between four and five of its `GroupedFeature` can be chosen. A `GroupedFeature` does not have `cardinality`. A `SolitaryFeature` is a feature that is not grouped in a `FeatureGroup`. The `cardinality` of `SolitaryFeature` specifies the maximum number of times this feature can appear in the final feature configuration. Figure 2 presents a feature model including features of the SPL of Smart Home systems.

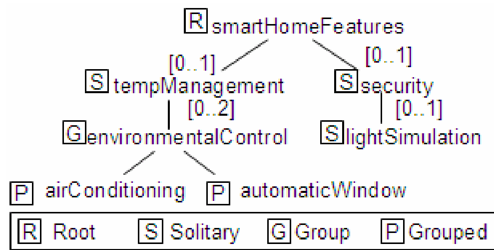

Fig. 2. Smart Home Feature Model.

Two `SolitaryFeatures`, `tempManager` and `security`, are at the root level. The `tempManager` feature has the `GroupFeature` `environmentalControl`, which has the `GroupedFeature` `airConditioning` and `automaticWindow`. The `security` feature has the `SolitaryFeature` `lightSimulation`.

3.2 Constraint Models

We call a *binding*, the materialization of a specific (fine-grained variation) choice for a product, expressed between elements of models and features. For example to express that, *the* `firstFloor` *has a central* `airConditioning` *that has to be turned on when the average temperature of the floor reaches* $22°C$, *and turned off when the temperature reaches* $18°C$.

Constraint models allow product line architects to restrict the bindings among elements and features that a product designer can establish. For example, to express that, *maximum three elements that conform to the metaconcept* `Window` *can be bound to the feature* `automaticWindow`.

A `constraint` $o = [M, F, P]$ is a tuple composed by a metaconcept M, a feature F, and a property P. A constraint o expresses the fact that model elements conforming to the metaconcept M can be bound to the feature F. We use the property P to define additional constraints that a model element has to satisfy to be bound to a feature. For example, the constraint `ConstraintAutoWindow`=(`Window`, `automaticWindow`, *cardinality-Temperature*) describes that, during the configuration of a specific Smart Home system, a product designer can bind elements conform to `Window`, for example the `mainRoomWindow`, to the `automaticWindow` feature. The property *cardinality-Temperature* specifies that, the number of elements that can be bound is minimum zero (0) and maximum three (3); and, bindings have to define the minimum temperature ($minTemp{>}0°C$) and maximum temperature ($minTemp{<}30°C$), to close and open respectively the automatic window.

3.3 Fine-Feature Configurations

Fine-feature configurations allow product designers to establish bindings among elements and features. Thus, while a constraint is expressed between metaconcepts and features, a binding is expressed between model elements and features. A binding $i = (m, F, R)$ is a tuple composed by a model element m, a feature F, and a logical expression R used to add semantic to the binding.

Figure 3 presents a fine-feature configuration with two bindings, `Binding1`= (`firstFloor`, `airConditioning`, {}), and `Binding2`= (`mainRoomWindow`, `automaticWindow`, R), where $R = \{maxTemp = 25\ and\ minTemp = 15\}$.

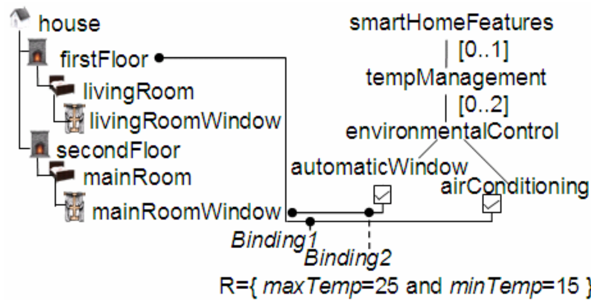

Fig. 3. Fine-Feature Configuration.

The `Binding2` expresses that the `mainRoomWindow` has an `automaticWindow` that has to be opened when the temperature of the room (`mainRoom`) reaches the maximum of $25°C$, and closed when the temperature reaches the minimum of $15°C$. To create constraint models and fine-feature configurations we have extended the Czarnecki et al. feature metamodel.

3.4 Bindings Vs. Constraints

We say that a binding $i = [m, F, R]$ *fits* a constraint $o = [M, F, P]$ when m conforms to M. We note this relationship $i \xrightarrow{f} o$. Therefore, since `mainRoomWindow`

conforms to `Window`, then `Binding2`$\xrightarrow{f}$`ConstraintAutoWindow`. We automatically validate fine-feature configurations against constraint models. This ensures that each binding i *fits* a constraint o, and satisfy the respective property P. Thus, fine-feature configurations are valid configurations inside the scope of the product line, representing products that include fine-grained variations.

4 Deriving Products

We configure products using feature models and fine-feature configurations. This allows us to express coarse- and fine-grained variations respectively. During the activity of product derivation, we execute successive *model transformations*, starting from domain models, until we derive particular products. We call a *workflow* the successive model transformations we execute. We call *model transformations*, programs that transform source models into target models by using sequences of *transformation rules*.

In our approach, we identify two types of transformation rules. We call *common-transformation rules*, the rules we use to generate the common characteristics of products; and, we call *fine-transformation rules*, the rules we use to generate the fine-grained variations of products. Thus, to generate a base product without including fine-grained variations, we execute a workflow including model transformations that use sequences of common-transformation rules. To generate a product including fine-grained variations, we adapt common-transformation rules in model transformations. Adaptations allow us to derive products including fine-grained variations. For instance, we can conditionally avoid the execution of a common-transformation rule, or to execute a fine-transformation rule *after* or *before* a common-transformation rule is executed.

A recent work [5, 7] has shown that model transformation programs can be modularly adapted based on AOP principles as introduced by AspectJ [1]. Using the same principles, we adapt model transformations by applying aspects to model transformations. An aspect specifies the common-transformation rules to intercept (pointcuts), and the fine-transformation rules to execute (advices).

We use *transformation interceptors* to intercept model transformations and apply aspects to common-transformation rules. We make conditional the execution of transformation interceptors by using *flow conditioners*. A flow conditioner determines when a transformation interceptor has to be executed, according to the current configuration (fine-feature configuration) of a product.

Figure 4 presents a workflow example (`SmartHomeWF`) that makes possible the derivation of Smart Home Systems based on a fine-feature configuration (`Fine-FeatureModel`).

The flow conditioner (`Conditioner`) checks if exist bindings in the `Fine-FeatureModel` that fit a specific constraint. If any, the transformation interceptor (`Interceptor`) is executed. `Interceptor` intercepts the model transformation `WindowsModelTransf`, which transforms `Window` elements, and execute the aspect `AutoWindowAspect`. The aspect specifies the pointcut, in this

case the common-transformation rule `Gen-TransRule-2`, and the advice, in this case the fine-transformation rule `F-AutoWindow`.

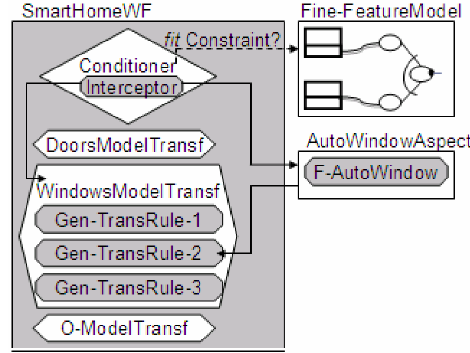

Fig. 4. Workflow Example.

4.1 openArchitectureWare

To implement our approach, we have used and extended the openArchitectureWare toolkit (oAW). oAW is a MDD framework integrated into Eclipse. oAW integrates facilities not only to transform models-to-models, but also to transform models-to-text (source code). At the core of oAW, there is a workflow engine allowing the definition of transformation *workflows* by sequencing diverse *workflow components*. oAW has some pre-built workflow components that facilitate the reading and instantiation of models, checking them for constraint violations, and transforming them into other models or source code. Transformation and generation workflows are built by using XML files that describe the steps needed to be executed in a generator run. Each of these steps is specified as a workflow component.

In oAW, model-to-model transformations are implemented using a language called Xtend. It is a textual functional language for querying and navigating existing models as well as building new models.

oAW provides support for AOSD. There is a workflow component that facilitates the definition of transformation interceptors. This is called the *transformationAspect* component. Furthermore, Xtend includes support for describing aspects that use *around* advices on transformation rules.

Listing 1 shows a workflow example. First, an EMF model (`domainModel.xmi`) is read. A workflow slot `source` is used to send the `domainModel` to the transformation component. Next, the model is transformed by invoking a model transformation (line 9) that starts with the execution of the transformation rule `createSimpleWindows`. The transformed model is available for further processing in the `target` slot.

```
1 <workflow>
   <component id='xmiParser'
3   class='org.openarchitectureware.emf.XmiReader'>
    <modelFile value='domainModel.xmi'/>
5   <outputSlot value='source'/>
   </component>
7
   <transform id='XtendComponent.model2model'>
9   <invoke value='createSimpleWindows(\${source})'/>
    <outputSlot value='target'/>
11 </transform>
  </workflow>
```

Listing 1. oAW Workflow Example

4.2 Loading Fine-Feature Configurations

To create a SPL including fine-grained variations, we create a general constraint model. We configure particular products including fine-grained variations creating fine-feature configurations. Using oAW, we create a workflow with scheduled workflow components that execute model transformations; and, we create common-transformation rules using Xtend.

Since we apply advices to common-transformation rules only in the case we find bindings that fit specific constraints, we need to query the fine-feature configuration at any moment of the oAW workflow execution. To perform this, we have created an oAW component that makes possible to load a constraint model and a fine-feature configuration into the execution context of an oAW workflow. Thus, a constraint model and a fine-feature model have to be available to be queried during the execution of the workflow. We call this component, the *fine-configuration loader* component.

4.3 Intercepting Workflows Execution

During the definition of oAW workflows, we use the oAW *transformationAspect* component to define transformation interceptors. As we introduced above, we make conditional the execution of transformation interceptors using flow conditioners. A flow conditioner indicates when a transformation interceptor has to be executed, according to the current fine-feature configuration loaded in the context of the oAW workflow execution.

We have created an oAW component to define flow conditioners. We call this, the *fine-feature* component. This component allows us to query a loaded fine-feature configuration to know if there are bindings that imply the adaptation of particular model transformations. For example, assume that we have a binding i such as, $i \xrightarrow{f}$`ConstraintAutoWindow`. In this case, a model transformation transforming elements that conform to `Window` must be intercepted. After the interception is done, the model transformation can be adapted to transform the elements involved in such bindings (elements that conform to `Window`) into `automaticWindows`.

Listing 2 shows a workflow example using the *fine-feature* component. The component queries a loaded fine-feature configuration searching bindings $i = (m, F, A)$ such that, m conforms to the `Window` metaconcept (line 4) and F is the feature `automaticWindow` (line 5). If any, the (oAW component) transformation interceptor `transformationAspect` (line 6) is executed. This indicates that the model transformation `XtendComponent.model2model` has to be intercepted (line 7), and the aspect `extensionAdvice` (line 8) has to be executed.

```
<workflow>
2  <component id='fineFeature'
    class='org.oAW.fineVariation.FineFeature'>
4    <isBoundAny value='Window'/>
    <toFeature value='automaticWindow'/>
6    <transformationAspect adviceTarget=
      'XtendComponent.model2model'>
8      <extensionAdvice value='extensionAdvice'/>
    </transformationAspect>
10 </component>

12 <transform id='XtendComponent.model2model'>
    ...
14 </transform>
</workflow>
```

Listing 2. Fine-Feature Component Example.

4.4 Using Fine-Transformation Rules

To create products including fine-variations, we adapt the execution of common-transformation rules by using aspects. We create aspects taking advantage of the facilities included in the (oAW) Xtend language. In particular, the pointcuts we define match execution points of common-transformation rules, and the advices we apply are fine-transformation rules.

Since Xtend facilitates the definition of *around* advices, once a common-transformation rule is intercepted, we can decide either to avoid its execution, or to execute it before or after our fine-transformation rule.

A fine-transformation rule transforms models to include fine-grained variations. This rules query fine-feature configurations to know the specific elements of a model, involved in bindings, that have to be transformed to include fine-grained variations. For example, assume the case where a $Binding2 =$ (`mainWindow`, `automaticWindow`,$\{\}$) $fits$ the `ConstraintAutoWindow` (see Figure 3). A fine-transformation rule queries the fine-feature configuration and identifies the `mainWindow` element. Then, the rule transforms the `mainWindow` into an `automaticWindow`.

Listing 3 presents an example of an Xtend aspect and a fine-transformation rule. In the aspect, the common-transformation rule `createSimpleWindows()` is intercepted (line 1) and the fine-transformation rule `fineRule()` is executed (line 2). Note that in this example, the common-transformation is avoided. Then, `fineRule()` executes a function `fit(String)`, which returns the model elements involved in bindings that fit the `ConstraintAutoWindow` (line 5). Finally, each of such elements is transformed into elements of the target model representing `automaticWindows` (line 6).

```
1 around createSimpleWindows():
    fineRule(f, res);
3
  private fineRule():
5  let windows = fit('ConstraintAutoWindow')):
   windows.createAutomaticWindow;
```

Listing 3. Example of Xtend Aspect and Fine-Transformation Rule.

5 Conclusions

In this paper, we introduced an approach to define fine-grained variations of SPLs while ensuring consistency between variation points using an aspect- model-driven approach. Our proposal is based on feature modeling, meta-modeling and AOP principles. We have used Domain Specific Modeling to address the problem of expressing SPL variability. Thus, we allow the easy configuration of products done by product designers who are experts in specific domains.

We have expressed fine-variations by using fine-feature configurations, and we have constrained such variations creating constraint models. Thus, we achieved to express fine-grained details of product configurations, included in a well defined scope of a SPL.

In SPL engineering, problem space focuses on defining what problem the family of applications, or an individual application in the family, will address. The solution space focuses on producing the software components to solve that problem. Our approach narrows the gap between the problem and the solution space deriving products by means of automatic model transformations. We derived products that include fine-grained variation crosscutting common-transformation rules with fine-transformation rules. For this, we have used AOP applied in the context of model transformations to derive products where variability has been bound to specific model elements.

References

1. AspectJ website: AspectJ website http://www.eclipse.org/aspectj.
2. Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
3. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. Volume 1241 of Lecture Notes in Computer Science. Springer-Verlag, New York, NY (jun 1997) 220–242
4. Third Workshop on Models and Aspects Handling Crosscutting Concerns in MDSD, ECOOP. (2007)
5. Voelter, M.: Patterns for Handling Cross-cutting Concerns in Model-Driven Software Development. In: 10th European Conference on Pattern Languages of Programs (EuroPLoP). (2005)
6. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2006)

7. Voelter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: Proceedings of the 11th International Software Product Line Conference. (2007) 233–242
8. C-SAW website: C-SAW website http://www.cis.uab.edu/gray/ Research/C-SAW/.
9. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. The Theme Approach. Addison-Wesley, Amsterdam (2005)
10. Groher, I., Voelter, M.: Expressing Feature-Based Variability in Structural Models. In: Workshop on Managing Variability for Software Product Lines. (2007)
11. Groher, I., Voelter, M.: Xweave: models and aspects in concert. In: AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling, New York, NY, USA, ACM (2007) 35–40
12. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In Glück, R., Lowry, M.R., eds.: GPCE. Volume 3676 of Lecture Notes in Computer Science., Springer (2005) 422–437
13. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models. In: Proceedings of the Third Software Product Line Conference 2004, Springer, LNCS 3154 (2004) 266–282
14. Garces, K., Parra, C., Arboleda, H., Yie, A., Casallas, R.: Variability Management in a Model-Driven Software Product Line. Avances en Sistemas e Informática **4**(2) (September 2007) 3–12
15. openArchitectureWare (oAW) website: http://www.openarchitectureware.org.
16. European Commission STREP Project AMPLE IST-033710: http://ample.holos.pt/.
17. Eclipse Modeling Framework (EMF) website: http://eclipse.org/emf.
18. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 (1990)

DiVA: Dynamic Variability in complex Adaptive systems

Dhouha Ayed

THALES
ThereSIS Security and Information Systems
Route départementale 128
91767 Palaiseau Cedex, France.
Dhouha.Ayed@thalesgroup.com

Abstract. The goal of DiVA is to provide a new tool-supported methodology with an integrated framework for managing dynamic variability in adaptive systems. This goal will be addressed by combining aspect-oriented and model-driven techniques in an innovative way

Keywords: MDA/MDE, aspect-oriented modeling, dynamic variability, adaptation to the context.

1 Introduction

Adaptive systems are able to dynamically adapt to their context. Managing adaptation of complex systems that include multiple heterogeneous interacting services is difficult since these services tend to be distributed, interdependent and sometimes tangled with other services. Furthermore, the exponential growth of the number of potential system configurations derived from the variability of each service needs to be handled. DIVA[1] is an European research project. Its objectives are:

1) To provide novel build-time and runtime management of adaptive systems having co-existing co-dependent configurations that can span across several administrative boundaries in a distributed, heterogeneous environment.

2) To provide efficient handling of the number of potential configurations, that may grow exponentially with each new variability dimension.

3) To increase quality and productivity of adaptive system development and help the designers to model, control and validate adaptation policies as well as the trajectory going from one safe configuration to another.

In order to achieve these objectives, we propose a new approach where we combine aspect-oriented and model-driven techniques. The paper is organized as follows. Section 2 describes the DiVA approach. Section 3 introduces the industrial

[1] http://www.ict-diva.eu/

case studies we use to evaluate the approach. Section 4 presents related work. Section 5 provides a conclusion.

2 Proposed Approach

The idea of the approach we propose is to combine model-driven and aspect-oriented techniques to handle the complexity of adaptive system construction and execution. Models cope with complexity through abstracting the dynamic variability. Aspect-oriented techniques are used to model the adaptation concerns separately from the other aspects of the system. By using model-based abstractions and advanced separation of concerns, the adaptation becomes easier to design and understand, possible to validate and allows to easily evolve the adaptation policies even at runtime. In the following sections, we show how we use the two techniques to manage dynamic adaptation.

2.1 Model-driven Approach for Dynamic Adaptation Management

The idea is to use models at two levels in order to manage dynamic variability: at design time and runtime (see Figure 1). In the following we specify how models are used at each level.

At design-time, a system is modelled using a base model which contains its basic functionalities and a set of variant models which can be composed with this base model. In order to identify which variant has to be selected depending on the context of the running system, an adaptation model is specified. The adaptation model specifies the following elements:

- A set of dependencies that specify constraints on variants. For example, the use of a particular variant might require or exclude others. These constraints reduce the total number of configurations by rejecting invalid configurations.
- A context model of the system
- A set of adaptation rules that link the context elements from the context model to the variants that should be used according to the context variation.

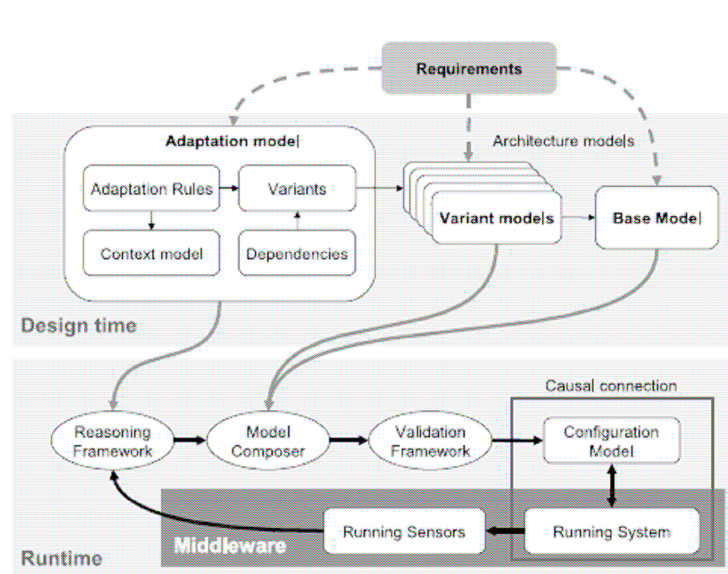

Fig. 1. Model-driven Approach for Dynamic Adaptation Management

At runtime the configurations of the system are built by selecting the variants that adapt to the context and composing them with the base. To select the configuration that adapts to the context, the reasoning framework collects context information from sensors and processes the adaptation model. The output of the reasoning framework is one or more options that match the adaptation rules and satisfies the dependency constraints. For each of these options the complete model of the corresponding configuration can be built at runtime using model composition (see more details in Section 2.2). The correctness of the composed configurations are checked with respect to the specified dependencies.

Once a configuration has been selected and checked, the running system can be adapted. The adaptation is carried out through a model that is causally connected to it. This model is transformed to match the configuration that has been selected. The running system is adapted thanks to the causal connection. The main advantage of using causally connected models for dynamic reconfiguration is the automatic generation of reconfiguration scripts instead of writing them by hand.

2.2 Aspect-oriented Modelling Approach for Dynamic Adaptation Management

We use aspect-oriented modelling techniques in order to tackle the issue of the combinatorial explosion of variants. Aspect-oriented modelling allows us to encapsulate distinct variation points into aspects which are separated from the base model of the system functionality. Then, distinct aspects might be composed into the base model in order to obtain different configurations. This approach allows us to

reason on a limited and linearly increasing number of aspects, thus avoiding the specification of all the possible configurations and consequently the problem of combinatorial explosion.

In practice, the variants are defined as aspect models to be woven with the base model. From a particular selection of variants, the corresponding configuration can be built automatically by weaving the corresponding aspect models with the base model.

3 Industrial Case Studies

Case studies from two different domains are used to validate the DiVA approach:

- The first case study is about the crisis management at an airport where the configuration of an airport system is dynamically adapted according to the crisis situation. A crisis can cause the failure of one or several critical systems of the airport, it can require a dynamic connection to external systems such as a police station, a hospital or a ministry, and it may need the reconfiguration of non-functional services such as the security management and the communication.
- The second case study is about the Customer Relationship Management (CRM) based on dynamic combination of services (service mash-ups) and service adaptation according to the user requirements and preferences.

4 Related Work

Several existing approaches focus on underlying mechanisms to support dynamic reconfiguration using reflection [1]. Considerable attention has been placed on policies and associated policy languages for reconfigurable systems, and on rule-based approaches and associated interpretation frameworks [2,3,4]. Significant work focused on autonomic computing leading to the concept of emergent behaviors in evolutionary systems [5]. In DIVA the runtime model uses variation points as a dynamic representation of variability. Thus, the reflection abilities are moved up to the model level. A key benefit is that models can be used to provide a richer semantic base for runtime decision-making related to system adaptation and other runtime concerns. In DiVA we can use models to help determine when a system should move from one consistent architecture to another and compute safe migration paths.

In MADAM and MUSIC[6,7] the main variability mechanism consists in loading different implementations for each component type of the architecture. DIVA reduces the complexity by decomposing the system into a base model and aspects.

5 Conclusion

DIVA proposes a novel combination of Model-Driven Engineering and aspect-oriented Modelling to manage dynamic variability of complex systems. Aspect-oriented Modelling allows to reduce the combinatorial explosion of variants. Model-Driven Engineering automates and improves the creation of the reconfiguration logic. During the next two years, we plan to elaborate our approach by validating it with the industrial case studies and developing a reasoning framework that will automatically select and weave the most adapted aspects. Using a causally connected model to manipulate the running system increases response times for adaptive systems. The evaluation of our approach efficiency is consequently necessary.

6 References

1. Coulson, G., Blair, G.S., Clark, M. and Parlavantzas, N. The design of a highly configurable and reconfigurable middleware platform. *ACM Distributed Computing Journal*, *15* (2). 109–126, April 2002.
2. McKinley, P.K., Sadjadi, S.M., Kasten, E.P. and Cheng, B.H.C. Composing Adaptive Software. *IEEE Computer*, *37* (7). 56-64, 2004.
3. Keeney, J. and Cahill, V., Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework. in *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, 2003.
4. Capra, L., Emmerich, W. and Mascolo, C., Reflective Middleware Solutions for Context-Aware Applications. in *Third International Conference on Meta-level Architectures and Separation of Crosscutting Concerns*, Japan, 2001.
5. Kephart, J.O. Research challenges of autonomic computing. in *27th international conference on Software Engineering (ICSE)*, St. Louis, MO, USA, 2005.
6. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., and Gjorven E., Using architecture models for runtime adaptability. in *Software IEEE*, 23(2):62–70, 2006.
7. Hallsteinsen, S., Stav, E., Solberg, A., and Floch, J., Using product line techniques to build adaptive systems. in *SPLC'06: 10th Int. Software Product Line Conf.*, pages 141–150,Washington, DC, USA, 2006. IEEE Computer Society.

GReCCo: Composing Generic Reusable Concerns*

Aram Hovsepyan, Stefan Van Baelen, Yolande Berbers, Wouter Joosen

K.U.Leuven DistriNet, Celestijnenlaan 200A, B-3001 Leuven, Belgium
{Aram.Hovsepyan, Stefan.VanBaelen, Yolande.Berbers, Wouter.Joosen}@cs.kuleuven.be

1 Introduction

In this paper we give a brief description of GReCCo, an Aspect-Oriented Modeling based framework to promote and enhance the reuse of concerns. GReCCo supports (1) *composition oblivousness* by modelling concerns independently from a concrete context in which they are going to be applied, (2) *composition symmetry* by treating all concerns (including the base concern) uniformly, and (3) *interdependency management* by a coupling to the Concern Interaction Acquisition (CIA) system. We have developed a prototype composition engine implemented in ATL that can be used to compose concern models specified in UML.

2 General description of GReCCo

GReCCo enables the composition of oblivious concern models. We represent each composition step as the Greek letter upsilon (ϒ). The left and the right branches of the upsilon contain two concern models. Our approach is composition symmetric in the sense that we treat *component* and *aspect* concerns uniformly. In order to combine the concern models, we provide a composition model that instructs the model transformation engine how the two models should be composed.

Concern Models describe the structure and the behavior of the concerns using respectively UML class and sequence diagrams. The *Composition Model*, which also consists of UML class and sequence diagrams using the GReCCo profile, specifies how the concern models are composed by defining all composition-specific parameters and their bindings. This assures a higher degree of reusability of the two concerns as they can be used in different contexts (*composition obliviousness*). The *Composition Model* describes how the source concern models should be composed, and is therefore by definition depending on these models. This allows to isolate the necessary links between the concerns into the composition model, thereby keeping the concerns oblivious and reusable. Using a generic composition engine, we generate the output *Composed Model* from the input composition and concern models.

* The described work is part of the EUREKA-ITEA SPICES project, and is partially funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders), by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

3 Composition Specification

In order to compose two concern models, we need to specify the composition by defining the *Composition Model*. Elements that are not directly referenced in the *Composition Model* are copied to the *Composed Model.* Other elements are modified by the composition engine according to the composition specification.

3.1 Structure

We distinguish five structural composition directives in total. From the point of view of a single concern model, we distinguish the following directives that involve one element: (1) we can *add* a new element, (2) we can *modify* the properties of an existing element, and (3) we can *remove* an existing element. When two concern models are composed, there are some additional composition directives that we can specify for the input elements. We can (4) *merge* two elements to obtain a single entity with the combined properties. Some concerns introduce roles and/or template parameters as semantic variation points, which we can (5) *instantiate* by using concrete UML elements.

3.2 Behavior

We use UML sequence diagrams to describe the behavior of concerns, which must be in correspondence with their structural counterpart specified by UML class diagrams. A concern model may contain several sequence diagrams. Each sequence diagram represents a certain scenario. Scenarios from the input concern models that are completely independent of each other, are simply copied to the composed model. We define two scenario's as independent of each other if they can be executed in parallel, meaning that the messages can be freely interleaved. For overlapping behavior scenarios we need to be able to address the following use-cases: (1) specify the sequence of messages between the input behavior scenarios, (2) indicate that a call from one input scenario is the same as a call from the other input scenario, and (3) replace a call or a set of calls from one input scenario by a call or a set of calls from the other input scenario.

To realize the composition, we introduce the UML2 notion of *general ordering*. A general ordering is a binary relation between two interaction fragments to describe that one of the fragments must occur before the other one. Each interaction fragment contains a set of events. The resulting scenario defines a partial ordering of the input events. The interpretation by the composition engine is that the dependency client fragment should immediately precede the dependency supplier fragment. Events that are not involved in any general ordering relation are put in parallel fragment blocks.

4 Future Work

In the future we plan to extend the GReCCo engine to support the behavioral composition and integrate it fully with the Concern Interdependency Acquisition (CIA) framework. We are also investigating the possibility to use domain-specific modeling languages for certain concerns. In addition, we plan to formalize the composition mechanisms and evaluate the scalability of our approach.