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.

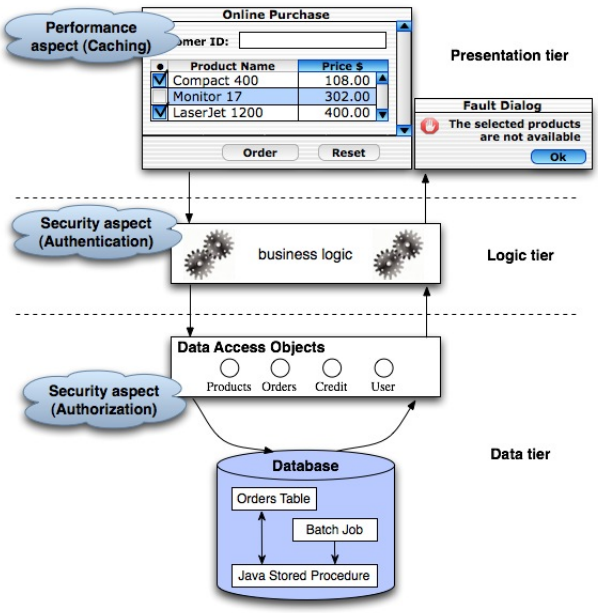

Fig. 1. Security and caching aspects apply at different layers of the architecture

– 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. For obvious reasons, it is preferable if the verification can occur statically, without executing the application under consideration.

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.

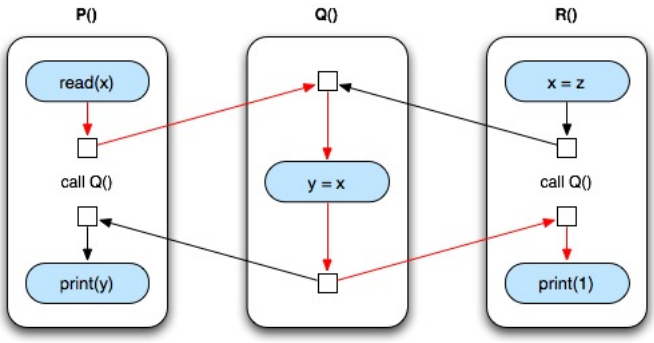

Fig. 2. Unrealizable paths when naively connecting control-flow graphs

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 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,

[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.

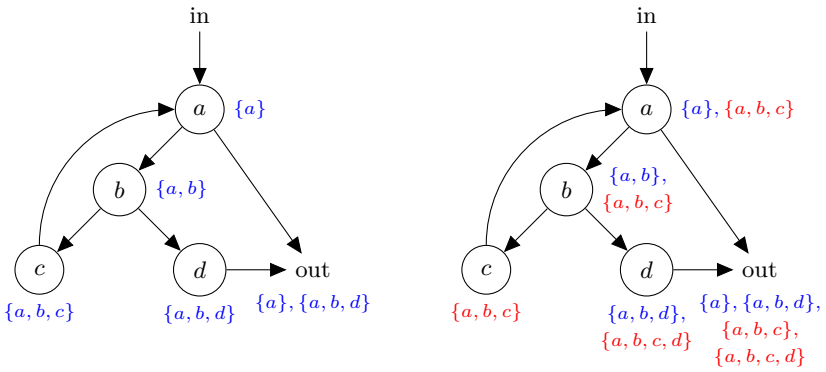

Fig. 3. Illustration of path calculation algorithm in first (l.) and second (r.) iteration

propagating the set of encountered nodes. This already produces two solutions 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)