Journal Subline

LNCS 5490

Transactions on Aspect-Oriented Software Development V

Awais Rashid · Harold Ossher
Editors-in-Chief

Springer

Lecture Notes in Computer Science 5490

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Awais Rashid Harold Ossher (Eds.)

Transactions on Aspect-Oriented Software Development V

Springer

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2, D.3, I.6, G.1.6, B.2.2, B.2.4, D.4.1

ISSN	0302-9743 (Lecture Notes in Computer Science)
ISSN	1864-3027 (Transactions on Aspect-Oriented Software Development)
ISBN-10	3-642-02058-5 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-02058-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12658091 06/3180 5 4 3 2 1 0

Editorial

Welcome to Volume V of Transactions on Aspect-Oriented Software Development. This volume is a combination of three papers submitted through the regular channel, and three papers submitted to the special focus area of *aspects, dependencies and interactions*. The first two regular papers are both on applications of AOSD to other fields: scheduling of web applications and operations research, while the third paper applies the technique of bisimulation to aspect-oriented languages.

The first paper, "Application-Level Scheduling Using AOP," by Kenichi Kourai, Hideaki Hibino, and Shigeru Chiba is a significant extension of their AOSD 2007 conference paper. They explore the use of aspects to facilitate application control over scheduling policies in web applications, achieving performance improvements without compromising modularity.

The second paper, "An Exploratory Study for Identifying and Implementing Concerns in Integer Programming," by Norelva Niño, Christiane Metzner, Alejandro Crema, and Eliezer Correa explores the application of AOSD to integer programming problems in operations research. They propose criteria for defining concerns in this domain, and report on the refactoring of the implementations of two algorithms in a well-known OR library to use aspects.

The third paper, "Open Bisimulation for Aspects," by Radha Jagadeesan, Corin Pitcher, and James Riely, extends their AOSD 2007 conference paper. It takes the coinduction technique, based on bisimulation principles, which has been successfully applied for proving program equality in other paradigms, and applies it in the context of aspects. The paper shows how bisimulation can be used as a mechanism for supporting expressive pointcuts for dynamic aspects.

The special focus area on *aspects, dependencies and interactions* was edited by guest editors Ruzanna Chitchyan, Johan Fabry, Shmuel Katz, and Arend Rensink, all well-known experts in this area, under the management of one of the co-editors-in-chief, Awais Rashid. This focus area is introduced in their guest editors' introduction. We wish to thank the guest editors for their effort and commitment in producing such a high-quality special focus area.

The next volume will be a special issue on *aspects and model-driven engineering*, with guest editors Jean-Marc Jezequel and Robert France. MDE is an important area of Software Engineering in which AOSD can play a significant role, so we eagerly anticipate publication of this collection of papers.

The AOSD field has produced a wealth of research of considerable diversity over many years now. There is, therefore, an important place for survey papers covering the field as a whole or important sub-areas. A good survey paper has a number of important elements, including clear and insightful exposition of the area covered, deep technical analysis of similarities and differences leading to useful concepts or abstractions, and good coverage of the work in the area with appropriate historical perspective and accurate coverage of key contributions. Such papers can be valuable introductions to researchers entering the field, or researchers in related fields trying to understand AOSD. We strongly encourage submission of survey papers to TAOSD,

and have adopted a different means of evaluating them that explicitly takes the criteria noted above into consideration.

As per the journal policy, the remaining founding co-editor-in-chief, Awais Rashid, is stepping down after the first four years of the journal. So this is the last volume Awais will be co-editing in this role. He has done a truly marvelous job of launching the journal successfully, together with his co-founding editor, Mehmet Aksit, and of steering its operations to date. He has been invaluable and always responsive in helping Harold become oriented to the job. The editorial board thank him most sincerely for his devoted service and his many contributions. Fortunately he will remain on the editorial board and continue to guide us.

At the same time, it is with great pleasure that we welcome Shmuel Katz, who will be taking over from Awais as co-editor-in-chief. Shmuel is well known both within and beyond the AOSD community, especially in the area of formal methods. His work on the calculus of superimpositions was a key, early contribution to the formal foundations of the AOSD field. We are privileged to have his guidance and leadership as we move beyond the startup phase of the journal.

February 2009

Awais Rashid
Harold Ossher
Co-Editors-in-Chief

Editorial Board

List of Reviewers

Table of Contents

Application-Level Scheduling Using AOP

Kenichi Kourai*, Hideaki Hibino**, and Shigeru Chiba

Tokyo Institute of Technology
{kourai,hibino,chiba}@csg.is.titech.ac.jp

Abstract. Achieving sufficient execution performance is a challenging goal of software development. Unfortunately, violating performance requirements is often revealed at a late stage of the development. Fixing a performance problem at such a late stage is difficult in terms of cost and time. To solve this problem, this paper presents *QoSWeaver*, which is a tool suite for developing application-level scheduling using aspects. QoSWeaver weaves scheduling code written in an aspect into web application code. The scheduling code gets an application thread to voluntarily yield its execution to implement a custom scheduling policy. The idea of scheduling at the application level is not new, but aspect-oriented programming (AOP) makes it more realistic by separation of scheduling code. For fine-grained scheduling, QoSWeaver provides a *profile-based pointcut generator*, which automatically generates appropriate pointcuts. To investigate the ability of QoSWeaver for implementing practical scheduling policies, we used QoSWeaver for tuning the performance of a river monitoring system named *Kasendas*, which is a web application system. For reliable examination, Kasendas was originally developed by an outside corporation and then it was tuned by the authors with QoSWeaver. The authors could successfully improve the performance of Kasendas under heavy workload. The cost of the performance tuning was reasonably small. Furthermore, our approach achieved better performance than other techniques such as admission control and priority scheduling provided by the JVM or Linux. We could implement various policies such as deadlock-aware or adaptive scheduling.

Keywords: scheduler, aspect, QoS, performance tuning, case study.

1 Introduction

Achieving sufficient execution performance is one of the primary goals of software development. However, it is always a challenging goal. For example, a web application may not satisfy its performance requirement but this fact is often uncovered when a stress test is performed at the final stage of software development or, in a worse case, after the application starts servicing to the users. Of

* Presently with Kyushu Institute of Technology.
** Presently with Hitachi Software Engineering Co., Ltd.

A. Rashid and H. Ossher (Eds.): Transactions on AOSD V, LNCS 5490, pp. 1–44, 2009.

course, the performance characteristics of the software should be carefully considered at the stage of architecture design but estimating the actual performance is difficult at that stage.

Fixing a performance problem at such a late stage is difficult in terms of cost and time. Some readers might think that the problem can be fixed by upgrading hardware, but this approach is the last resort because it needs extra cost. In the case of web applications, the second-best solution would be to improve the quality of service (QoS), but it is still a challenge. To exploit schedulers provided by operating systems or middleware for controlling QoS, developers must modify web applications, sometimes largely. Such modification may be difficult to finish within a limited time. If the scheduling policy provided by operating systems or middleware is not suitable for the web applications, developers can use a different operating system or middleware. However, changing such underlying software requires them to test their software again because they must guarantee that the software system correctly works under the new circumstances. Executing all the test again would take a long time.

To solve this problem, this paper presents *QoSWeaver*, which is a tool suite for developing application-level scheduling using aspects. QoSWeaver enables changing a scheduling policy for web applications on demand. QoSWeaver weaves scheduling code written in an aspect into application code. The scheduling code gets an application thread to voluntarily yield its execution to implement a new scheduling policy. The idea of scheduling at the application level is not new, but aspect-oriented programming (AOP) makes it more realistic by separating scheduling code from applications. AOP prevents application logic from being corrupted when scheduling code is added or changed. This has been one of the major obstacles to adopt application-level scheduling. In addition, QoSWeaver provides a *profile-based pointcut generator* which helps developers write aspects for fine-grained scheduling. The pointcut generator automatically generates pointcuts so that the scheduling code is executed at as regular intervals as possible, according to profile information of the execution of web applications.

To examine that QoSWeaver enables implementing a practical non-toy scheduling policy, we used a river monitoring system named *Kasendas*, which is a web application that periodically collects the water levels of major Japanese rivers and reports the collected data to the public through the web. We then executed the performance tuning of Kasendas so that it can periodically collect water levels at correct intervals even if a large number of clients simultaneously send requests to visualize the data of water levels. From the viewpoint of thread scheduling, we tried to give sufficient CPU time to the thread for periodically collecting water levels than the other threads for processing requests from clients. For reliable examination, we ordered the initial development of Kasendas to an outside corporation and we only executed performance tuning. We used QoSWeaver and we could successfully implement a scheduling policy that gives sufficient CPU time to the thread for collecting water levels. The work of the performance tuning was not large compared with the modification of the software design of Kasendas.

Our contributions presented in this paper are the following:

- We propose an AOP-based implementation of application-level scheduling, which is a new application from the AOP perspective.
- We report our experience of applying the application-level scheduling to a fairly realistic web application.
- We show that our approach worked well at least in our case study.

Note that this paper does not propose a new scheduling or resource-allocation algorithm. It proposes using AOP for implementing an application-specific scheduler. Since AOP makes the implementation easier, the use of such a custom scheduler is made practical. Although this paper deals with QoS, we implemented proportional-share scheduling; it is not real-time scheduling. It is out of the scope of this paper whether or not the proposed approach can be used to implement a real-time scheduler.

This paper is an extended version of our previous paper [1]. A difference between the two papers is that this paper presents two more scheduling policies implemented by our QoSWeaver: deadlock-aware and adaptive scheduling. Another difference is that this paper also shows that our application-level scheduler achieved better performance than the priority schedulers included in the standard Java virtual machine (JVM) and the Linux kernel. This paper also shows that the use of our pointcut generator makes a non-negligible impact on the overall performance. Selecting appropriate pointcuts is significant from the performance viewpoint.

The rest of this paper is organized as follows. Section 2 explains why fixing a performance problem at a late stage of software development is difficult and describes related work. Section 3 presents QoSWeaver, which enables application-level scheduling by using AOP. Section 4 illustrates a river monitoring system named Kasendas, which is our case study, and shows an applied scheduling policy. Section 5 reports the results of our experiments to examine the usefulness of QoSWeaver. Section 6 discusses the applicability of QoSWeaver. Section 7 concludes this paper.

2 Motivation

A web application normally processes various kinds of tasks requested from web browsers (i.e. users) in parallel. Some kinds of tasks have higher importance while others have lower importance. The QoS of such a web application is often kept acceptable if higher-importance tasks obtain more computing resources such as CPU time than lower-importance tasks. However, this solution is still a challenge. Since modern operating systems provide a scheduling mechanism for controlling QoS, some readers might think that what developers should do is only to slightly modify their web applications to exploit that scheduling mechanism. Unfortunately, in reality, it is not such a simple thing.

First of all, the software sometimes has to be largely modified to exploit that scheduling mechanism. Such modification is not easy to finish within a

limited time before the expected shipping date. For example, to use real-time scheduling provided by operating systems, developers may have to move a part of application code into kernel modules. Even if developers just want to use priority scheduling, which is provided by most operating systems, and raise the priorities of some threads, they may change their applications so that the threads will run with the root privilege. This change may cause security problems and thus the developers must check the entire code of their applications. In addition, developers cannot exploit the scheduling mechanism of the operating system if an application (Java) thread is not always bound to a particular operating-system thread. This depends on the implementation of the application threads. If the mapping between application threads and operating-system threads may change, for example, it is difficult to raise the priority of a particular application thread by changing the priorities of operating-system threads.

Furthermore, scheduling policies provided by operating systems may not be suitable for web applications. For example, the priority scheduling provided by some general-purpose operating systems may not allocate sufficient CPU time to an application thread executing a periodic task with a high priority. If there are too many low-priority threads, a high-priority thread tends to miss its deadline. This problem will be avoided if developers use a different operating system, in particular, a real-time operating system, but changing an operating system at the final stage of software development is not acceptable. According to our previous work, the performance behavior of web applications largely changes if the underlying operating system is changed, even from a general-purpose one to another [2]. Developers must spend a long time for testing an entire software system again. They must guarantee that the software system correctly works under the new circumstances.

Exploiting the QoS mechanism provided by middleware has a similar problem. The standard JVM supports priority scheduling of Java threads, but it does not guarantee its effectiveness. Priorities passed from applications to the JVM are only used as hints. Whether or not the priorities really affect the scheduling depends on the implementation of the JVM and the underlying operating system. If the scheduling policy provided by the standard JVM is not suitable, developers can use another implementation of the JVM, for example, a real-time JVM implementing Real-Time Specification for Java [3]. However, changing the JVM at the final stage is not acceptable as well as changing the operating system. Although some web application servers provide built-in mechanisms for controlling QoS, those mechanisms are often insufficient. For example, if the maximum number of threads is limited to avoid excessive concurrency, high-priority threads cannot start execution when too many low-priority threads are already running.

2.1 Aspect-Oriented QoS Control

Re-QoS [4] uses a QoS aspect package to adapt applications to the real-time systems. A QoS aspect package is a set of aspects and components that provide different QoS policies. In their case study, the authors showed that Re-QoS could control the deadline miss ratio by admission control of requests. Although

Re-QoS uses aspects to add a new QoS management policy like our QoSWeaver, it is difficult to use Re-QoS for fine-grained scheduling because the developers have to find appropriate pointcuts by hand. On the other hand, our QoSWeaver provides the pointcut generator, which automatically generates pointcuts so that scheduling code will be executed periodically.

QuO [5] builds a QoS management system as an aspect and weaves it into the boundary between the application and the middleware. Its aspect language allows the developers to describe adaptive QoS, which changes the behavior of applications according to available system resources. QuO supports distributed object middleware like CORBA and it compiles an aspect into a delegate, which is a proxy for calls to remote objects (remote method invocation in Java). Therefore, QoS control is enforced only when an application calls remote objects. This is not sufficient for applications that do not frequently call remote objects.

Bossa [6] enables a scheduling expert to implement a new scheduling policy for operating system kernels. It provides a domain-specific language (DSL) to describe a scheduling policy using high-level abstractions. This DSL simplifies scheduler programming and allows the formal verification of safety properties. To make the kernel raise scheduling events to a scheduler compiled as a kernel module, Bossa inserts the code for raising events into the kernel using AOP [7]. Using DSL and AOP, Bossa allows web application developers to change the kernel scheduler without changing the source code of the operating system. However, if the developers change the kernel scheduler, they need to spend a long time for examining the scheduling behavior of the entire software system.

2.2 Previous Approaches to Application-Level Scheduling

MS-Manners [8] achieves process scheduling called progress-based regulation at the application level. It stops low-importance processes when the progress rate is lower than expected and it gives remaining CPU time to high-importance processes. Its platform-independent implementation is to insert calls to the Testpoint function everywhere in a program. This function examines the progress rate and blocks the process for a while if necessary. This method is similar to our approach. However, MS-Manners requires the developers to manually modify the source code of their applications so that Testpoint will be called. They may also have to modify the source code of the libraries used by their applications. Otherwise, the developers have to write scheduling code together while they are writing the application logic. Either way, their productivity will be decreased. Maintaining the source code becomes difficult because the scheduling code is directly embedded into the source code. To solve these problems, QoSWeaver separates scheduling code into an aspect and it automatically inserts scheduling code at appropriate places in the source code when it performs weaving.

For UNIX processes, several application-level scheduling mechanisms have been proposed. User-level scheduling [9] and ALPS [10] control UNIX processes from a scheduler process by using signals such as SIGSTOP and SIGCONT. User-level sandboxing [11] restricts the CPU usage of processes by changing the priorities of threads. These mechanisms enable more accurate control than QoSWeaver

because they can perform preemptive scheduling. However, it is difficult to apply them to threads instead of processes. User-level scheduling and ALPS distinguish applications by running them with different user accounts. User-level sandboxing enforces a policy by a controller attached to a process. These mechanisms cannot distinguish or control threads. In addition, preemptive scheduling is difficult to implement on the standard Java 2 Platform. Its API specification does not recommend to use APIs for suspending and resuming threads by another thread. Also, Java provides the API for changing the priorities of Java threads but its effectiveness is not guaranteed.

Gatekeeper [12] can transparently apply admission control and request scheduling to servers by using a proxy server. The admission control limits the number of concurrently processed requests to avoid excessive concurrency. The request scheduling changes the order of handling requests to improve the response time. The proxy analyzes the kinds of requests and schedules the requests appropriately. Installing the proxy does not need modifying operating systems, middleware, or applications. However, if there are heavy-weight applications, which take a long time for processing each request, the threads for those applications are not controllable once they start the execution. They cannot be suspended to decrease concurrency. QoSWeaver enables finer-grained control by weaving scheduling code, for example, at a method-call granularity.

Admission control based on the SEDA architecture [13] enables fine-grained scheduling by dividing an application into several stages [14]. In SEDA, the execution of the application is performed by sending a request to the next stage. Each stage has a queue to receive the request and allows admission control for each request. If an application is divided into a sufficient number of small stages, fine-grained scheduling is achievable. The advantage of this architecture is that there are no threads suspended by a scheduler unlike our approach. Until a request is admitted, no thread is allocated to it. However, the developers must re-implement their applications using multiple stages to fix performance problems if they have already implemented the applications.

3 Aspect-Oriented Application-Level Scheduling

To solve the problem described in the previous section, this paper presents QoSWeaver, which is a tool suite for developing application-level scheduling by aspects. It enables developers to customize a policy of thread scheduling at the application level. In this section, we describe how AOP makes application-level scheduling feasible in practice.

3.1 Application-Level Scheduling

Application-level scheduling is implemented by the cooperation among application threads, which voluntarily yield their execution in favor of other threads. Thus, a thread must periodically invoke a method on a scheduler object. The scheduler's method suspends the caller thread according to a specified scheduling

policy. The suspended thread can be woken up and rescheduled when another thread calls the scheduler's method. The scheduler's method suspends a caller thread only if the *yield flag* of the thread is set. Thus, we can control the scheduling by setting and clearing this flag. Suppose that a scheduling policy is that all other threads are suspended while a particular thread A is running. This policy is implemented as illustrated in Fig. 1. If the thread A first calls the scheduler's method, the method does not suspend the thread but sets the yield flag of another thread B. This will suspend the thread B when the thread calls the scheduler's method next time. The thread B will not be woken up again until the thread A finishes its execution and the scheduler clears the yield flag of the thread B.

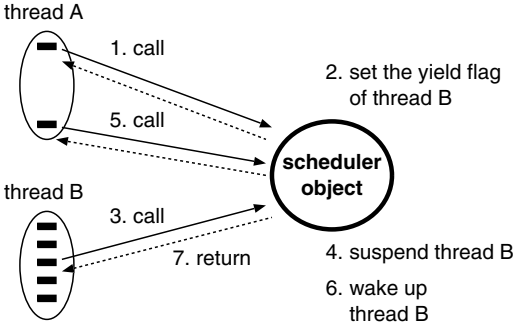


Fig. 1. A scheduling method based on thread yielding

Application-level scheduling has several advantages compared with scheduling at a lower level such as the operating system level or middleware level. One advantage is to enable developers to implement various scheduling policies without modifying the underlying systems. Application-level scheduling is independent of the underlying operating system and middleware and hence it does not need to change them. It changes only the scheduling policy of the target applications. Since application-level scheduling affects only the threads of the target applications, the rest of the threads in the software can obtain at least the same amount of CPU time as they can when application-level scheduling is not applied. Of course, application-level scheduling cannot achieve all kinds of scheduling independently of the underlying schedulers. For example, time-sharing scheduling cannot be developed on top of a batch scheduler. Another advantage is to enable developers to develop application-specific schedulers. Such schedulers can use application semantics given by application threads. For example, if application threads inform a scheduler of their roles, the scheduler can allocate CPU time to these threads according to their role. If a low-level scheduler is used, application threads must translate such high-level semantics to low-level properties such as thread priorities.

3.2 AOP Support for Application-Level Scheduling

The idea of scheduling at the application level is not new but it has not been practical because the developers have to insert scheduling code into their application programs by hand. For example, our web applications presented in Sect. 4 consist of about 10000 lines of our own code and more than 100000 lines of library code. It is difficult to insert scheduling code at right places, in particular, for average developers. Hence the code they inserted must be later checked by an experienced developer. This work is annoying and time consuming. Moreover, if a scheduling policy requires a thread to frequently yield its execution, developers must insert scheduling code at a large number of places. This causes the application logic to be tangled with scheduling code. Maintaining the tangled scheduling code is difficult. For example, if developers want to change a scheduling policy, they may have to remove old scheduling code and insert new scheduling code. This modification is error-prone and hence developing an appropriate scheduling policy by a trial-and-error approach is difficult.

QoSWeaver lets developers to write scheduling code as an aspect and weaves it into application code. AOP makes the idea of the application-level scheduling practical. Since scheduling code is separated from application-logic code, it can be written by only a few experienced developers. Other average developers do not have to write scheduling code any more and can concentrate on writing application-logic code without being aware of scheduling. Writing scheduling code as an aspect also allows developers to take a trial-and-error approach to develop an appropriate scheduling policy. Developers can easily try various scheduling policies to find the most appropriate one because an aspect weaver automatically inserts and removes scheduling code. They never accidentally corrupt their programs when they change a scheduling policy.

Furthermore, QoSWeaver supports creating a scheduling mechanism for application-level scheduling. Scheduling code written as an aspect consists of a scheduling mechanism and the implementation of a scheduling policy. A scheduling mechanism for application-level scheduling is a set of method calls on a scheduler object from various places in application programs as in Fig. 1. This corresponds to timer interrupts for kernel-level scheduling. A scheduler for kernel-level scheduling is called periodically from hardware. A scheduler for application-level scheduling is called from the joinpoints selected by pointcuts. AOP is used to implement this mechanism.

A *pointcut generator* provided by QoSWeaver automatically generates a set of pointcuts to create such an application-specific scheduling mechanism. This tool helps developers define a right set of pointcuts for getting an application to call a scheduler at as regular intervals as possible. Calling a scheduler at regular intervals is desirable for stable control of application threads. In particular, fine-grained scheduling needs such a tool support because an application needs to frequently call a scheduler to yield its execution. It is difficult to manually define pointcuts for fine-grained scheduling because the pointcuts must select a large number of joinpoints and a thread must reach those joinpoints in regular intervals. Furthermore, the number of the joinpoints selected by pointcuts should be

minimum; otherwise, a scheduler will be called redundantly. Calling a scheduler twice within a single interval is useless. The second call is just a performance penalty.

Figure 2 illustrates the architecture of QoSWeaver. QoSWeaver consists of two tools: an AOP system and a pointcut generator. QoSWeaver receives a web application and weaves a profiling aspect, which is provided by QoSWeaver, into it using the aspect weaver. Then, QoSWeaver deploys the extended web application for profiling on a web application server. If developers run it, the profiling aspect records its execution profile. From the execution profile that the aspect recorded, the pointcut generator generates appropriate pointcuts for efficient application-level scheduling. Then, developers write a scheduler aspect by using the pointcuts. A scheduling policy is implemented as advice, which is executed at scheduling points specified by the pointcuts. In Fig. 1, it is implemented by the scheduler object. Finally, QoSWeaver weaves this aspect into the original web application and deploys the resulting web application on the server.

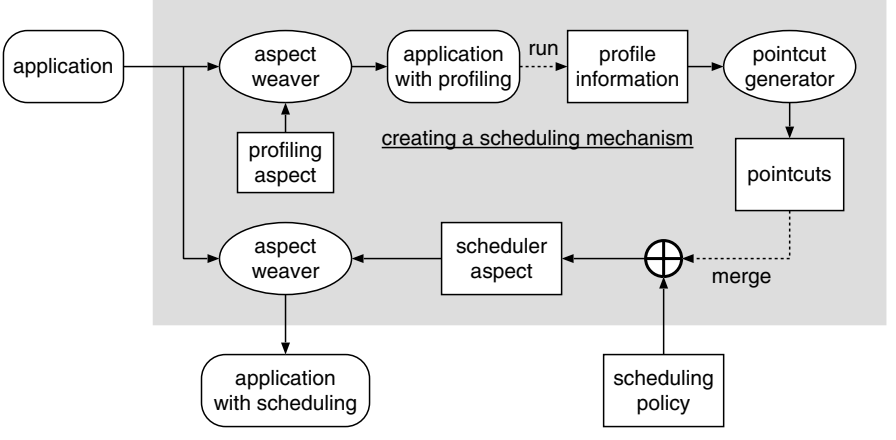


Fig. 2. The architecture of QoSWeaver

Note that QoSWeaver does not directly support the development of a scheduling policy itself. It only generates appropriate pointcuts for creating a custom scheduling mechanism. Details of a scheduling policy have to be chosen by developers. For example, how priorities are assigned to tasks depends on the user requirements for the web applications. The maximum number of threads concurrently running for each task depends on scheduling algorithms. These parameters should be selected by experiments. QoSWeaver helps developers to select those parameters by using a trial-and-error approach.

3.3 Profile-Based Pointcut Generator

The pointcut generator generates appropriate pointcuts on the basis of the profile information of the execution of a target application. The profile data collected by

QoSWeaver is when a thread reaches each joinpoint. Our current implementation of the pointcut generator deals with only method calls as joinpoints. QoSWeaver first weaves a target application with a profiling aspect that records a caller's method name, a callee's method signature (the method name, the parameter types, and the return type), and the time stamp for each method call. Then, developers run the target application into which the profiling aspect has been woven. Since their application is a web application, they also run a client to send requests to the application. The client sequentially sends requests to minimize the disturbance by the outside because we want to know when each single thread calls which method.

The execution profile should be the representative of the behavior of a target web application. If the behavior of an application is largely different from the execution profile, the application could not call the scheduler at regular intervals. However, it is difficult to guarantee that the obtained execution profile is a representative because this fact strongly depends on the characteristics of applications. A guideline for obtaining a representative execution profile is to send the most common request to the target web application. The developers should know what the most common request is for their applications. If the behavior of the application largely changes by request patterns, they can obtain multiple execution profiles for all the patterns and merge generated pointcuts. Currently, QoSWeaver does not provide any support for obtaining a representative execution profile.

To generate appropriate pointcuts from that execution profile, the pointcut generator takes two parameters from the developers:

- a *target interval* between adjacent joinpoints selected by pointcuts, and
- the acceptable *maximum occurrences* of joinpoints selected by a single pointcut.

The target interval specifies the time from when an application thread calls a scheduler until when the thread calls it again, on the profiled execution. The criterion for the pointcut generator is that the average time elapsed between adjacent joinpoints selected by pointcuts is close to the target interval t given from the developers. The pointcut generator generates pointcuts that satisfy this criterion as much as possible. The maximum occurrence m is used to avoid that too many joinpoints are selected.

These two parameters should be determined so that QoSWeaver will generate a scheduling mechanism with acceptable accuracy and overhead for an intended scheduling policy. The accuracy and overhead of application-level scheduling also depends on the characteristics of the web application, the patterns of requests to web applications, and the underlying systems. Therefore, developers should give different sets of parameters to the pointcut generator and obtain multiple sets of pointcuts. Then they should choose the best one by changing the sets of pointcuts and running the web application for evaluation. Our pointcut generator makes this kind of parameter tuning easy.

Note that the target interval is the interval to be achieved in the same execution contexts as when we obtained the execution profile. During the profiled

execution, we ran only one application thread on a server. If production run is also single-threaded, an observed interval, at which an application thread calls a scheduler, would be almost the same as the target interval on average. If it is multi-threaded, however, an observed interval for each application thread would be longer than the target. In application-level scheduling, the observed interval largely depends on the patterns of the requests to web applications.

Algorithm for Pointcut Generation. Figure 3 shows our algorithm of pointcut generation. The inputs to this algorithm are an execution profile and the two parameters described above. The execution profile consists of the data of the joinpoints that an application thread reached during the profiled execution. In our implementation, it is a sequence of the invoked method calls, as in Fig. 4. According to the time stamps recorded at the joinpoints, the algorithm divides the sequence into time slots by the target interval t. The aim of this algorithm is that the generated pointcuts select only one (or as a small number as possible) joinpoint for each time slot. Since a scheduler is called only at selected joinpoints, this algorithm enables application threads to call a scheduler at as regular intervals as possible. The generated pointcuts are chosen among possible pairs of call and withincode pointcuts. The call pointcut specifies a method by its name and signature and matches points at which the method is called at run time. The withincode pointcut limits the scope in which the call pointcut selects method calls to a specified method body. Thus, each pair of pointcuts selects joinpoints representing calls to the same method within the same method body. In this algorithm, no pointcut includes wildcards.

The algorithm first chooses pairs of pointcuts that select a joinpoint occurring only once in the execution profile. For example, if a method A is called from a method B only once during the entire profiled execution, the caller and the callee are used to make a pair of withincode and call pointcuts. Let PC_1 be the set of chosen pointcuts. Then, the algorithm computes a subset of PC_1 that covers as many time slots as possible. Here, covering a time slot means that a joinpoint selected by a pointcut occurs in that time slot. If there are multiple pointcuts that cover the same time slot, the algorithm chooses one of them. Let PC_{gen} be the computed subset of PC_1.

Then, for the time slots that have not been covered, which are denoted by $SLOT$, the algorithm chooses pointcuts that select joinpoints occurring only twice in the execution profile. Let PC_2 be the set of chosen pointcuts. The algorithm computes a subset of PC_2 that covers as many not-covered time slots as possible. If there are multiple pointcuts that cover the same time slot, the algorithm chooses the pointcut that covers the most time slots. The elements of the computed subset are added to PC_{gen}. If PC_{gen} contains an element that covers the same time slots as an element newly added to PC_{gen}, then the former element is removed from PC_{gen}. If the former element covers a time slot that the latter element does not cover, it is not removed. The algorithm iterates this choosing process from PC_1 to PC_m, where m is a parameter given by the developers. After the iteration, the pointcut generator generates PC_{gen} as its result.

$t := target\ interval$
$m := maximum\ occurrence$
$exec\_time := total\ execution\ time$

$PC_{all} := a\ set\ of\ possible\ pointcuts$
$PC_{gen} := \{\}$
$SLOT := \{0, ..., \lceil exec\_time/t \rceil\}$

for each $i = 1..m$
  $PC_i = \{pc \in PC_{all} \mid |select(pc)| = i\}$
  for each $j \in SLOT$
    $PC_{ij} = \{pc \in PC_i \mid j \in cover(pc)\}$
    $PC_{best} = \{pc \in PC_{ij} \mid |cover(pc) \cap SLOT|\ is\ biggest\}$
    $best\_pc = eval(PC_{best})$
    $PC_{del} = \{pc \in PC_{gen} \mid cover(pc) \subset cover(best\_pc)\}$
    $PC_{gen} = PC_{gen} - PC_{del} + best\_pc$
    $SLOT = SLOT - cover(best\_pc)$
  endfor
endfor

Fig. 3. The algorithm for pointcut generation. Function *select* receives a pair of call and withincode pointcuts. It returns a set of joinpoints selected by the pair. Function *cover* receives a pair of pointcuts and returns a set of time slots covered by the pair. Function *eval* receives a set of pairs of pointcuts and returns one of them. $|S|$ is the size of a set S.

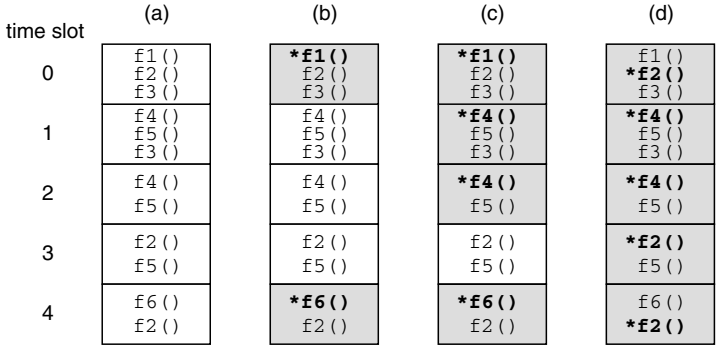


Fig. 4. An example of pointcut generation. A sequence of method calls invoked during profiled execution is divided by a target interval. Marked method calls are selected joinpoints.

For example, suppose that the profiled execution is modeled as a sequence of invoked method calls, f1() to f6(), in Fig. 4a. For simplicity, we ignore caller's methods in this example. The profiled execution consists of five time slots. The algorithm first chooses a pointcut that selects f1() in time slot 0 and one that selects f6() in time slot 4. Now, these two pointcuts are in PC_{gen} and the time slots 0 and 4 are covered (Fig. 4b). Then, the algorithm chooses PC_2, which

contains two pointcuts: a pointcut that selects two f3() and one that selects two f4(). Since the pointcut that selects f4() covers two new time slots (time slots 1 and 2), the algorithm adds the pointcut that selects f4() to PC_{gen} (Fig. 4c). The pointcut that selects f3() covers only one new time slot (time slot 1). At the final iteration, a pointcut that selects three f2() is added to PC_{gen}. At the same time, the pointcuts that select f1() and f6() are removed from PC_{gen} (Fig. 4d). The time slots 0 and 4 are covered by the pointcut that selects f2(). The algorithm results in the pointcuts that select f2() and f4(). In this example, all the time slots are covered by those pointcuts and each time slot includes only one selected method call.

3.4 Concerns for Scheduling Policies

Synchronization. Scheduling code woven into applications by QoSWeaver includes synchronization code for suspending and restarting a thread. We implemented the synchronization by the Object.wait and Object.notify methods in Java. Adding such synchronization code may cause deadlocks if the original applications also include synchronization code. Suppose that threads A and B running in an application access the same synchronized object in a synchronized block as shown in Fig. 5. If joinpoints in the block are selected by pointcuts, the thread B calls the scheduler object and can be suspended in the block to yield the allocated CPU time. While the thread B is suspended in the block, the thread A will be blocked if it attempts to enter the block because the thread B does not release the lock. If the thread A has a role to wake up the thread B within or after the block, a deadlock occurs. The thread A cannot wake up the thread B forever. However, typical web applications do not often include synchronization code. In particular, the Enterprise JavaBeans (EJB) specification [15] prohibits using thread synchronization. In fact, the web applications that QoSWeaver was applied to in Sect. 4 did not use thread synchronization although they were not EJB applications.

If web applications include synchronization code, developers can write scheduling policies that wake up suspended threads periodically and run application code a little. This is implemented by using the Object.wait method with timeout in scheduler code. As shown in Fig. 5, when the specified timeout is expired after thread B was suspended, thread B executes the scheduler code, which checks the progress of the other running threads, in this case, thread A. If some threads do not make progress for a while, the scheduler decides to restart thread B temporarily because those threads may make no progress due to deadlocks. If all threads make progress when thread B calls the scheduler code at the next joinpoint selected, thread B calls the Object.wait method to suspend again. It is guaranteed that the applications recover from a deadlock if all the suspended threads temporarily run, as far as the original applications do not include deadlocks in their logic. This approach also prevents livelocks, that is, a restarted thread never suspends instantly again for waiting the same lock. When a suspended thread is restarted by timeout, it necessarily proceeds to the next joinpoint selected by pointcuts.

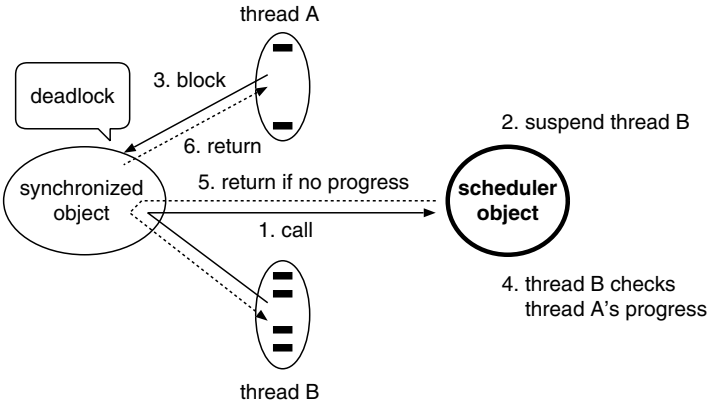


Fig. 5. A scheduling method for breaking deadlocks

To record the progress of threads for the above method, AOP is also useful. For example, QoSWeaver can weave an aspect for incrementing a progress counter at various joinpoints in applications. Such joinpoints should be selected by pointcuts so that a thread will reach them at as regular intervals as possible and the accurate progress can be monitored. Our pointcut generator can generate appropriate pointcuts for that purpose.

If applications are written without using thread synchronization, the developers can write scheduling policies concisely because they do not need to consider deadlocks. Even when thread synchronization is used, deadlocks are avoidable if the usage is localized. The pointcut generator can generate pointcuts that do not select any joinpoints in synchronized blocks.

Some readers may think that using an independent scheduler thread is straightforward to solve deadlocks because the scheduler thread can always run. However, it is difficult in Java that such a scheduler thread preemptively suspends other threads. Although Java provides the Thread.suspend and Thread.resume methods for thread preemption, these methods are not recommended to use because they are inherently deadlock-prone. If an application thread is suspended within a synchronized method in a system class, the scheduler thread may block at that method and then a deadlock may occur.

I/O. When a thread blocks for I/O, the schedulers of the underlying operating system and middleware automatically allocate CPU time to another thread. An application-level scheduler does not need to reschedule threads whenever a thread issues blocking I/O. This makes it easy to implement scheduling policies. This mechanism assumes that the underlying schedulers schedule threads in a fair manner. This assumption is satisfied in most operating systems and middleware. If developers want to control threads strictly, they can modify scheduling policies to make a thread call a scheduler and temporarily run another thread just before it issues blocking I/O. After the thread completes the I/O, it can immediately call the scheduler to suspend the temporarily running thread.

4 Case Study

To examine that QoSWeaver enables implementing practical scheduling policies, we executed performance tuning of a web application system with QoSWeaver. The web application that we used is a river monitoring system named Kasendas. This section describes the overview of the web application and what scheduling policy we developed for the web application.

4.1 Kasendas: A River Monitoring System

Kasendas is a river monitoring system that collects and reports the water levels of major rivers in Japan to the public through the web. Figure 6 shows a screenshot of its client's view. The web applications that supply such information related to natural disaster should be carefully implemented to be able to work under a large number of simultaneous accesses, known as *flash crowds*. Usually people will not access such a web application but, once a large typhoon approaches, they will rush to the web application for making sure that their local rivers are not flooded. We executed performance tuning so that the software will work under such heavy workload. We chose this application because this work was done for demonstrating our AOP technology within the framework of a research project funded by Japan Science and Technology Agency, which is studying dependable IT infrastructure for secure life. Since Kasendas is for technology demonstration, the water levels shown by Kasendas were pseudo data produced by the data generator, which emulates sensor nodes that measure the water levels of rivers and provides the data to Kasendas.

To make the results of our experiment reliable, Kasendas was initially developed by an outside corporation with CMMI level 3 [16]. We only received its source files and executed performance tuning. Although we told them the aim

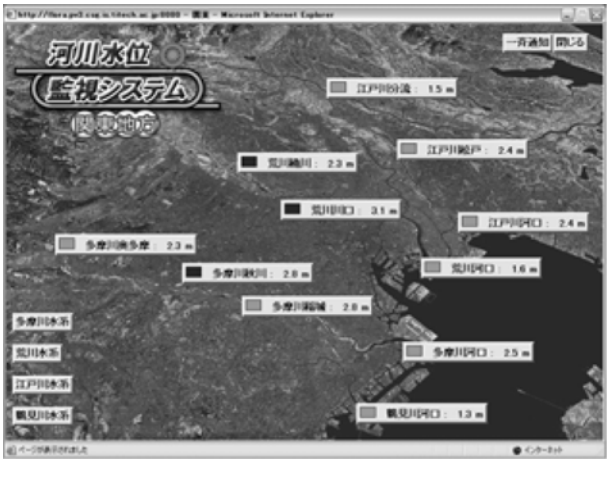


Fig. 6. The up-to-date water levels in Tokyo shown by Kasendas

Table 1. The code size of Kasendas

Server	File type	Number	Lines
Kasendas	.java files	82	9238
	JSP files	12	1736
	dicon files	15	558
Data generator	.java files	8	646

of Kasendas, they developed it independently of QoSWeaver. The requirement from us was to build Kasendas with typical open source middleware: JBoss application server [17], the Tomcat web container [18], the Struts framework [19], and the Seasar2 container [20]. Table 1 shows the code size of Kasendas. JSP files specify the design of web pages and the dicon files specify the configuration of components. This table does not include third-party libraries and frameworks. The development cost of Kasendas was 8.8 man-month, including tests and the design of web pages.

Figure 7 shows the architectural overview of Kasendas. Kasendas has three web applications for *periodic data collection*, *chart generation*, and *water-level update*. One web application collects the water levels of rivers through web services provided by the data generator periodically, for example, every 15 s. To collect the water levels of all rivers, the application sends the same number of requests as rivers to the data generator. The collected data are stored in the PostgreSQL database [21] and the latest data are also kept in memory. The other two applications generate web pages for the users. One generates a web page showing recent changes of water levels, for example, for the last 12 h. It reads data from the database and generates a chart of water levels by using the JFreeChart library [22]. This is a heavy-weight application because it accesses the database and produces a PNG image of the chart like Fig. 8. The other generates a web page showing up-to-date water levels. It reads data on memory and generates an image like Fig. 6.

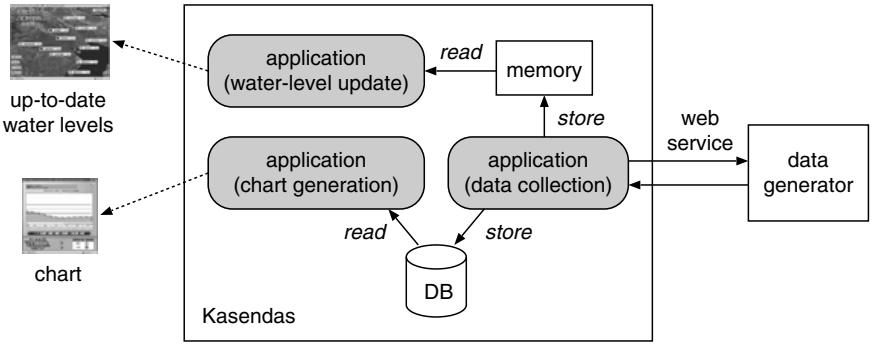


Fig. 7. The architecture of Kasendas

Fig. 8. The generated chart of recent changes of water levels in a river

Kasendas executes these three applications as follows. A timer in Kasendas triggers the execution of the application for periodic data collection. A single thread allocated by the timer executes the application at regular intervals. On the other hand, the other two applications for the chart generation and the water-level update are executed when Kasendas receives requests from the clients. Since these applications must be able to process a number of simultaneous requests in parallel, they are multi-threaded. When Kasendas receives a request, it obtains a thread from the thread pool provided in the web application server and the thread executes the requested application.

The initial version of Kasendas that we obtained from the outside corporation was unstable under heavy workload. It frequently failed to collect water levels on time from the data generator. According to our investigation, it became unstable when a number of clients simultaneously access the web page showing a chart. Since generating that page is a heavy-weight task, it consumes a large amount of CPU time and thus it disturbs another application for periodic data collection. This collector application will miss its deadline and lose a part of the water levels at that time. Furthermore, this application continues to collect the rest of the water levels in the next time period because it is not aware of the deadline miss. Thus, it fails again to collect the up-to-date water levels in the next time period.

4.2 The Applied Scheduling Policy

To fix this performance problem, we used QoSWeaver. There were two performance requirements:

- preventing a deadline miss in the periodic data collection even under heavy workload, and
- preventing performance degradation of the chart generation.

Note that true real-time scheduling was not required for this case. These requirements are somewhat vague and they are about user experiences. Our goal is to satisfy these requirements *as much as possible* with a minimum software development cost when we already have a program that mostly works well on a normal software stack.

Therefore, the scheduling policy applied to Kasendas was proportional-share scheduling for two groups of threads for the chart generation and a thread for periodic data collection. The former threads have low importance and the latter has high importance in this policy. While the high-importance thread does not run, the scheduling policy runs all the low-importance threads to generate a chart. When the high-importance thread starts running for periodic data collection, the scheduling policy quickly limits the number of low-importance threads to keep the ratio of the number of threads for each group. The scheduling policy did not consider threads that execute the application for the water-level update. They did not have high importance and did not make the system load high because they were lightweight.

The scheduling policy makes a low-importance thread call a scheduler periodically. If the yield flag of the thread is set by the scheduler, the thread is suspended. On the other hand, the scheduling policy makes a high-importance thread call the scheduler when the thread starts periodic data collection. At this time, the scheduler limits the number of running low-importance threads by setting the yield flags. We experimentally configured the number to one, which made the behavior of the high-importance thread the stablest. This means that our scheduling policy gives a share of 50 % to each group of threads. We did not limit the number to zero because we wanted low-importance threads to run while a high-importance thread was running. The scheduling policy makes a high-importance thread call the scheduler again when it finishes the execution. The scheduler removes the limitation on the number of running low-importance threads, resets the yield flags of the low-importance threads, and wakes up all the suspended low-importance threads.

To write an aspect that implements this scheduling policy for QoSWeaver, we used our own AOP system named GluonJ [23]. In GluonJ, an aspect is written in XML as glue code.[1] An aspect is woven into web applications when they are loaded into a web application server. The aspect we wrote is shown in Figs. 9 and 10.

Figure 9 shows the part of pointcut declaration in our aspect. For simplicity, we omit package names from class names. A pointcut is declared within the pointcut-decl tag. It is named with the name tag and specified with the pointcut tag. In our case, we declared three pointcuts: lowImportance, highImportance, and controlPoint. The lowImportance and highImportance pointcuts consist of the execution pointcut, which specifies a method by its name and signature and matches points at which the method is executed at run time. The lowImportance pointcut selects the execution of the method starting the chart generation. The

[1] In the latest version of GluonJ, an aspect is written in Java. See http://www.csg.is.titech.ac.jp/projects/gluonj/.

```
<pointcut-decl>
   <name> lowImportance </name>
   <pointcut>
      execution(void PlaceChartCreatePseudActionImpl.execute(..))
   </pointcut>
</pointcut-decl>

<pointcut-decl>
   <name> highImportance </name>
   <pointcut>
      execution(void CollectorImpl.doObtain())
   </pointcut>
</pointcut-decl>

<pointcut-decl>
   <name> controlPoint </name>
   <pointcut>
      (withincode(Range CategoryPlot.getDataRange(ValueAxis)) ANDAND
       call(Range Range.combine(Range,Range)))
      OROR
        :
   </pointcut>
</pointcut-decl>
```

Fig. 9. The pointcut declaration in our aspect

highImportance pointcut selects the execution of the method starting periodic data collection. These two pointcuts were written by hand with the knowledge of the source code of Kasendas.

On the other hand, the controlPoint pointcut was generated by our pointcut generator. The definition of controlPoint consists of 17 pairs of withincode and call pointcuts, which are concatenated with ORORR. ANDAND and ORORR correspond to && and || in AspectJ [24], respectively. The call pointcut specifies a method by its name and signature and matches points at which the method is called at run time, and the withincode pointcut limits a caller's method within which the call pointcut matches method calls. The controlPoint pointcut selects various method calls during the chart generation as far as these methods are called from the specified caller methods. A scheduling mechanism specific to Kasendas is constructed from these pointcuts. The pointcuts specify scheduling points in the source code of Kasendas and its application threads call a scheduler implemented as advice when they reach the scheduling points at run time.

Figure 10 shows the part of advice declaration in our aspect. Advice is declared within the behavior tag. It consists of the pointcut tag and the around or before tag. The pointcut tag specifies a named pointcut declared with the pointcut-decl tag. The around and before tags specify around advice and before advice, respectively. The around advice is executed in place of its joinpoints selected by a pointcut while the before advice is executed before its joinpoints. In the around tag, proceed() calls the original method selected by a pointcut. A special

```
<behavior>
   <pointcut> lowImportance() </pointcut>
   <around>
      PSScheduler.startLowImportance();
      $_ = proceed($$);
      PSScheduler.endLowImportance();
   </around>
</behavior>

<behavior>
   <pointcut> highImportance() </pointcut>
   <around>
      PSScheduler.startHighImportance();
      $_ = proceed($$);
      PSScheduler.endHighImportance();
   </around>
</behavior>

<behavior>
   <pointcut> controlPoint() </pointcut>
   <before> PSScheduler.yield(); </before>
</behavior>
```

Fig. 10. The advice declaration in our aspect

variable $$ represents arguments passed to a target method and $_ represents a return value in this context. The advice bodies invoke the methods declared in the PSScheduler class, which is a support class of our aspect, shown in Fig. 11. Advice and its support classes are the implementation of a scheduling policy, which is named a scheduler.

The first around advice is executed when a low-importance thread generates a chart. It calls the scheduler to register the thread to be controlled before the selected method execution and to unregister it after that. The second around advice is executed when a high-importance thread performs the periodic data collection. It calls the scheduler to control the number of running low-importance threads before and after the selected method execution. This policy temporarily reduces the number to one and restores it to 40, which was equal to the maximum number of concurrent requests to a web page showing a chart in our experiments. While the periodic data collection was not performed, we did not need to restrict the number of running threads. The last before advice is executed before a low-importance thread performs the selected method calls during the chart generation. It calls the scheduler to yield the execution of the thread itself that executed the advice.

The ThreadController class used in PSScheduler implements our scheduling algorithm. This class is reusable and the code size is 151 lines. The implementation is as follows:

– The add method puts the current thread into a run queue of a scheduler if the number of threads in the run queue is under the configured maximum.

```
public class PSScheduler {
   private static ThreadController controller =
      ThreadController.getInstance();

   public static void startLowImportance() {
      controller.add(Thread.currentThread());
   }
   public static void endLowImportance() {
      controller.remove(Thread.currentThread());
   }
   public static void startHighImportance() {
      controller.schedule(1);
   }
   public static void endHighImportance() {
      controller.schedule(40);
   }
   public static void yield() {
      controller.yield(Thread.currentThread());
   }
}
```

Fig. 11. A support class for our aspect

Otherwise, the method puts the current thread into a wait queue and suspends the current thread by invoking the Object.wait method.

- The remove method removes the current thread from the run queue. If the number of threads in the run queue is under the maximum, the method resets the yield flags of some threads in the wait queue and wakes up the threads by invoking the Object.notify method.
- The schedule method moves some threads in the run queue to the wait queue if the number of threads in the run queue is above the new maximum specified by the argument. Then, the method sets yield flags of those threads. Otherwise, the method moves some threads in the wait queue to the run queue, resets their yield flags, and wakes up those threads.
- The yield method suspends the current thread, if its yield flag is set, by invoking the Object.wait method.

We did not specify a timeout for the Object.wait method because the original Kasendas did not include synchronization code among threads and it was guaranteed that suspended threads were always woken up by other threads.

4.3 Our Experiences

Throughout the development of our scheduling policy, we found that QoSWeaver made the development easy. First, our scheduling policy was not affected by the modifications of the source code of Kasendas, thanks to aspects and our pointcut generator. During 1 month before the final release of Kasendas, we had to develop the scheduling policy for the intermediate version of Kasendas

in parallel while the development team of Kasendas was still testing and fixing bugs. This is because we had to demonstrate Kasendas at a symposium held by our grant sponsor soon after the expected final release date. Since our scheduling policy was implemented by an aspect and its support classes, we could apply our scheduling policy to the final version of Kasendas without manual modifications of the policy. Although pointcut declaration strongly depends on the code of Kasendas, the pointcut generator automatically generated a new appropriate controlPoint pointcut for the final version.

Second, an aspect allowed us to change a scheduling policy without affecting the source code of Kasendas. We developed the best scheduling policy in the following steps. At the beginning, we tried to cause low-importance threads to yield their execution by getting them to sleep during a certain period by the Thread.sleep method. We implemented the scheduling policy that got threads to sleep at joinpoints *except* the JFreeChart library. This policy could not control a system load well because the execution of low-importance threads took a long time in that library. Next, we changed this policy so as to get threads to sleep at joinpoints *within* the library. This policy almost worked well, but it sometimes failed to collect water levels at correct intervals when many threads were woken up accidentally at the same time. Finally, we changed this policy so as to use the Object.wait method for thread yielding. This policy could always control thread execution properly. While we modified our aspect and its support classes through these steps, we could not need to modify the source code of Kasendas. In addition, it was easy to change our scheduling policy to other ones for our experiments described in the next section.

Third, our pointcut generator enabled us to select the controlPoint pointcut for periodic thread yielding without examining the source code of Kasendas in detail. For periodic thread yielding, we had to choose pointcuts that selected joinpoints that a thread reached at reasonable intervals. However, there were too many candidates for pointcuts in Kasendas even if we limited pointcuts to the pair of the withincode and call pointcuts without wildcards. For a web application generating a chart, there were 803 candidates of pointcuts even in the execution profile we obtained. If we did not have the execution profile, there were much more candidates in the source code of Kasendas. It was impossible to select appropriate pointcuts among these enormous candidates by hand. Using our pointcut generator, we only needed to run a target application for obtaining execution profile, which was used to run the pointcut generator with several sets of parameters, so that the best set of generated pointcuts would be experimentally selected.

The development of our scheduling policy was less than one man-month. Our student, who is one of the authors of this paper, found the condition where Kasendas became unstable and developed the best scheduling policy by trial and error. He found the condition in 1 week, developed a scheduling policy in less than 2 weeks, and tested and modified it in 1 week. For comparison, the developers of Kasendas proposed 0.9 man-month for modifying Kasendas for potential performance improvement. Note that the proposed work was not equivalent

to our work. It did not include the analysis of performance bottlenecks. The work was only modifying Kasendas to use multiple threads for collecting water levels in parallel from the data generator. Furthermore, the developers could not guarantee that their modification prevented data loss under heavy workload because they did not know the real performance bottlenecks. Since their modification would make the software more complicated, estimating the performance was difficult.

4.4 Additional Scheduling Policies

Deadlock-Aware Scheduling Policy. To examine that QoSWeaver can break deadlocks introduced by woven scheduling code, we modified Kasendas so that it would cause a deadlock when it ran with our scheduling policy described in Sect. 4.2. We added a Logging class including two synchronized methods to Kasendas. These two methods, writeCollection and writeGeneration, are called by a high-importance thread for the periodic data collection and low-importance threads for the chart generation, respectively. When a low-importance thread is suspended within the writeGeneration method and waits for being woken up by the high-importance thread, a deadlock occurs if the high-importance thread calls the writeCollection method. Since the high-importance thread blocks at the writeCollection method, it cannot wake up the suspended low-importance thread.

An aspect as shown in Fig. 12 records the progress of the high-importance thread. The advice calls the PSScheduler.setProgress method 100 times during the execution of the high-importance thread. The pointcut was generated by our pointcut generator on the basis of the execution profile for the periodic data collection. The setProgress method increments the value of a progress variable. In addition, we modified the controlPoint pointcut in our original aspect so that a joinpoint within the writeGeneration method was selected to yield thread's execution.

To enable breaking deadlocks, we also modified the yield method in the PSScheduler class. Within the yield method, a thread calls the Object.wait method with the timeout of 200 ms if its yield flag is set. When the execution of the

```
<pointcut-decl>
   <name> progress </name>
   <pointcut>
      (withincode(int CollectorImpl.getWaterLevel(int)) ANDAND
       call(int Integer.parseInt(java.lang.String)))
   </pointcut>
</pointcut-decl>

<behavior>
   <pointcut> progress() </pointcut>
   <before> PSScheduler.setProgress(); </before>
</behavior>
```

Fig. 12. The aspect added for measuring a progress

```
<pointcut-decl>
   <name> midImportance </name>
   <pointcut>
      execution(ActionForward AbstractMapAction.execute(..))
   </pointcut>
</pointcut-decl>

<behavior>
   <pointcut> midImportance() </pointcut>
   <around>
      PSScheduler.startMidImportance();
      $_ = proceed($$);
      PSScheduler.endMidImportance();
   </around>
</behavior>
```

Fig. 13. The aspect added for the middle-importance task

Object.wait method finishes, the thread checks its yield flag again. If the flag is reset, the thread finishes the execution of the yield method and executes application code because this means that the thread is woken up by the Object.notify method. Otherwise, the thread compares the current value of the progress variable with the saved previous one to check the progress of the high-importance thread. If the value is changed, the high-importance thread is running and no deadlock occurs. Therefore, the thread calls the Object.wait method again. Otherwise, it temporarily executes application code to break potential deadlocks. During this temporary execution, the yield flag is set.

After that, the low-importance thread checks the progress of the high-importance thread whenever the yield method is called. If the yield flag is set and if the high-importance thread makes no progress, the thread skips the rest of the yield method and continues the temporal execution. If the high-importance thread makes progress, the low-importance thread finishes the temporal execution and executes the yield method normally.

Adaptive Scheduling Policy. So far, we considered only the low-importance threads for the chart generation and the high-importance thread for the periodic data collection. Kasendas has another web application for the water-level update. This application is of intermediate importance and should be run by a middle-priority thread. It is more important than that for the chart generation because, in the disaster case, the up-to-date water levels are more critical information than their changes in the past. It is not as important as the application for the periodic data collection. When we consider such middle-importance threads as well, they conflict with both low- and high-importance threads. The throughput of the middle-importance task may decrease due to the low-importance task when many low-importance threads are running because the chart generation performed by the low-importance threads is a heavy-weight task. While the periodic data collection is performed, the time needed for periodic data collection

may be increased by the middle-importance task. Although the web application for the water-level update is lightweight, too many requests to the web application affect the other applications.

To minimize such conflicts, we extended our scheduling policy described in Sect. 4.2. The extended policy guarantees the target throughput for the middle-importance task on average when a high-importance thread does not run. It adaptively adjusts the number of low-importance threads so that the middle-importance task achieves the target throughput. While the high-importance thread is running, the policy limits the maximum throughput of the middle-importance task. It directly adjusts the execution of the middle-importance threads because the number of the low-importance threads is limited to one during data collection by our original policy and it is difficult to be adjusted furthermore. Figure 13 is an aspect added for the middle-importance task. The midImportance pointcut selects the execution of the method in the AbstractMapAction class, which is invoked from the ActionServlet class used by the Struts framework [19]. The around advice is executed when a middle-importance thread performs the water-level update.

First, we consider only middle- and low-importance tasks when the periodic data collection is not performed. To calculate the throughput of the middle-importance task, our scheduler counts the number of pages generated during a certain period. The number is incremented by the PSScheduler.endMidImportance method, which is called after the execution of the water-level update. To adjust the maximum number of low-importance threads, we added the adjust method to the ThreadController class, which is described in Sect. 4.2. The method is called in the startMidImportance, endMidImportance, and yield methods of the PSScheduler class. Thus, the adjust method is frequently called if the middle- or low-importance threads are running.

The adjust method adjusts the maximum number of low-importance threads if the specified time elapses since the last adjustment. It does nothing until the specified time elapses. In the current implementation, the time is 1 s. The `adjust` method first calculates the latest throughput of the middle-importance task from the number of generated pages and the time elapsed since the last adjustment. Then it calculates the ratio of the observed throughput to the specified target throughput. If the ratio is less than one, the method decreases the maximum number of low-importance threads by the ratio to decrease the load by the low-importance threads. If the ratio is more than one, the method increases the number by the ratio to suppress the execution of the middle-importance threads. For example, suppose that the maximum number of low-importance threads is 5. When the observed throughput is 180 pages/s and the target is 150 pages/s, the ratio is 1.2 and then the maximum number is increased to 6.

Next, we consider three kinds of tasks when the periodic data collection is performed. To limit the number of requests processed in parallel, our scheduler maintains the number of running middle-importance threads. The startMidImportance method increments the number and the endMidImportance method decrements it. While the high-importance thread is running, the

startMidImportance method checks the number of running middle-importance threads. If the number is more than the specified value, the current thread is suspended. When a middle-importance thread finishes the water-level update, the endMidImportance method wakes up one of the suspended threads. The woken-up thread waits for the specified time because of adjusting the rate of request processing. The maximum throughput is determined by the maximum number of middle-importance threads and the waiting time. When the high-importance thread finishes the periodic data collection, all the suspended threads are woken up. At the same time, the maximum number of low-importance threads is restored to the value just before starting the periodic data collection. This behavior is changed from the original policy, which restores the number to the fixed value, 40. This change enables quickly stabilizing the maximum number of low-importance threads after the periodic data collection.

5 Experiments

Our application-level scheduler successfully improved the execution performance of Kasendas. Interestingly, we could not achieve this improvement by using existing schedulers of the underlying software layers such as the JVM and the Linux operating system. This section illustrates this fact through the results of our experiments.

5.1 Five Versions of Kasendas

We ran not only our Kasendas tuned with QoSWeaver but also the original Kasendas without any tuning and other versions of Kasendas tuned with admission control, Java priority scheduling, and Linux priority scheduling. Admission control is a simple scheduling technique for limiting the number of threads concurrently running. Because of its simplicity, it is often used for controlling the concurrency of web application servers. A web application server adopting admission control checks the number of running threads when it receives a new request from a client. If the number exceeds the limit, the server does not start processing the new request. A main difference between admission control and our scheduling by QoSWeaver is that admission control can suspend processing a request only when the server starts processing it. Once it starts processing, a thread processing the request is not suspended until it finishes processing. It is never suspended halfway.

The admission control for Kasendas restricts the maximum number of running low-importance threads for generating a chart. It limits the maximum number to one while a high-importance thread is collecting water levels. This policy is the same as the policy of our scheduling except that it is enforced only when a thread starts. Thus, the comparison between admission control and QoSWeaver will reveal a performance benefit of enforcing the policy by suspending a thread halfway through the execution.

The other two versions of Kasendas were tuned by scheduling mechanisms in the JVM and the operating system. To support our claim in Sect. 2, it is

important to show that only using such existing scheduling mechanisms is insufficient. Java provides the Thread.setPriority method for priority scheduling. Using this method, Kasendas sets the priority of the high-importance thread to high (`MAX_PRIORITY`) while it sets the priority of the low-importance threads to low (`MIN_PRIORITY`). After the low-importance threads finish to process requests, Kasendas resets their priorities to normal (`NORM_PRIORITY`) because those threads were reused via a thread pool.

Kasendas tuned with Linux priority scheduling issues the setpriority system call using a native method in Java when threads start processing requests. Like the setPriority method in Java, Kasendas sets the priorities of the high- and low-importance threads to high (-20) and low (19), respectively. However, Kasendas has to be run with root privilege to raise the priority of the native thread. It has to change the priority of the high-importance thread from normal (0) to high. In addition, it has to change the priority of the low-importance thread from low to normal when it returns the low-importance thread to the thread pool.

We did not change the operating system and the JVM used in Kasendas to real-time ones because that is not realistic at software development in practice as described in Sect. 2. If we largely change such underlying software, we may rewrite our applications. In addition, all of the underlying software need to be changed to real-time systems, but there existed no real-time JBoss server, which Kasendas required.

For our experiments, the interval at which Kasendas collects water levels was 15 s. To generate workloads, we used Apache JMeter [25]. JMeter concurrently sent requests to the web page showing a chart of recent changes for the last 12 h, except one experiment in Sect. 5.2. The number of concurrent requests was 40, except one experiment in Sect. 5.3. The number, 40, was the maximum number of concurrent requests through our experiments because the chart generation was a heavy-weight task and processing 40 requests in parallel caused overload in our server. Although more requests may be sent to the server in practice, we assume that more than 40 requests are discarded by admission control. We did not send requests to the web page showing the up-to-date water levels, except one experiment in Sect. 5.5.

To run Kasendas and the data generator, we used two Sun Fire V60x with dual Intel Xeon 3.06 GHz processors, 2 GB of memory, a gigabit Ethernet NIC. These machines ran Linux 2.6.8 as the operating systems, Sun JVM 1.4.2_06, and JBoss 4.0.2 as the J2EE servers. To run JMeter, we used Sun Fire B100x with a single AMD AthlonXP-M 1.53 GHz processor, 1 GB of memory, and a gigabit Ethernet NIC. This machine ran the Solaris 9 operating system and Sun JVM 1.4.2_05. These machines were connected with a gigabit Ethernet switch.

5.2 Effectiveness of Our Scheduling

We examined whether our scheduling could give sufficient CPU time to the thread executing the application periodically collecting water levels.

Time for Collecting Water Levels. We measured the elapsed time from when a high-importance thread starts collecting water levels until it completes the collection. Since this data collection is performed periodically, data loss occurs if the collection does not finish within its interval, which was 15 s in our configuration. Our aim is to prevent such deadline misses for the periodic data collection.

Figure 14 shows the changes of the time needed for collecting water levels every 15 s and Fig. 15 shows the average collection time. When we used the original Kasendas, we could measure the collection time only six times during 180 s. This is because each data collection took long time. The average collection time was 24.8 s and every collection time was more than 15 s, which was a deadline, except at 30 s. On the other hand, our scheduling reduced the average collection time to 5.3 s. The collection time was always within 15 s and no data were lost. In addition, the collection time was the stablest among the five versions of Kasendas. The variance of our scheduling was the smallest. This is very important for applications with deadlines because it becomes easier to guarantee that applications do not miss their deadlines. For the admission control, the average collection time was 10.9 s, but the collection time sometimes exceeded 15 s, for example, at 30 s after the start. This means that the admission control could not always prevent data loss. Fine-grained scheduling by QoSWeaver could prevent data loss by giving sufficient CPU time to the thread for collecting water levels.

Kasendas tuned by Java priority scheduling achieved a little shorter collection time on average than the original one, but the average was still longer than 15 s. The average collection time of Kasendas tuned by Linux priority scheduling was 9.0 s. However, Linux priority scheduling was not as good as our scheduling. The collection time was not stable and longer than 15 s at the first data

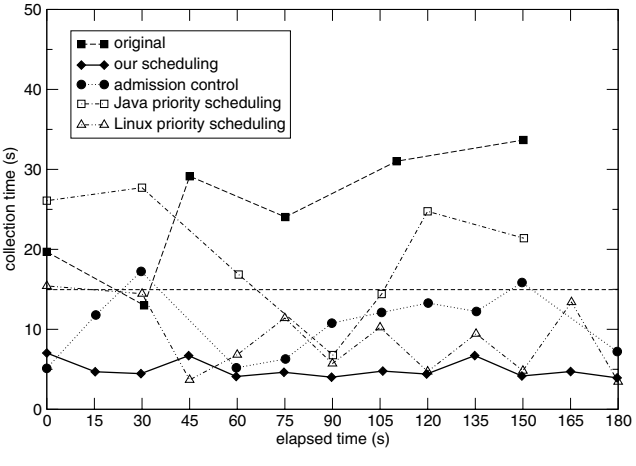


Fig. 14. The changes of the time needed for a high-importance thread to collect water levels

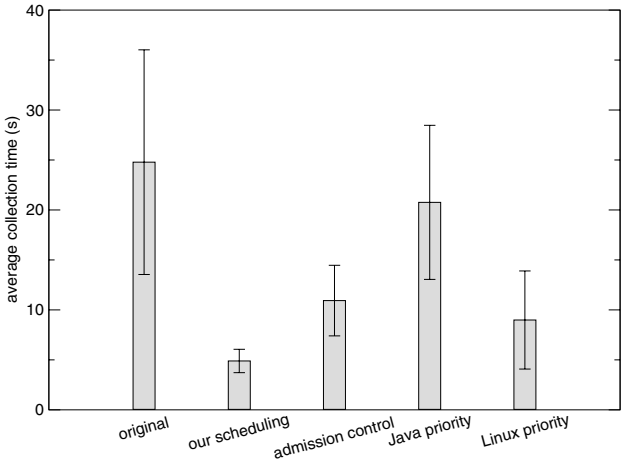


Fig. 15. The average time needed for a high-importance thread to collect water levels

collection. The cause of this instability is that the execution of the chart generation has several phases. Since JMeter simultaneously sent 40 requests at time 0, many threads tended to execute the same phase synchronously. Therefore, the characteristics in each phase affected the performance of the periodic data collection.

The deadline miss ratio was 0.89 for the original Kasendas, but it was reduced to zero by our scheduling. The other approaches could not achieve this: 0.31 for the admission control, 0.89 for Java priority scheduling, and 0.18 for Linux priority scheduling.

Number of Running Low-Importance Threads. To examine the scheduling behaviors in detail, we measured changes of the number of running low-importance threads for generating a chart. In our configuration, both our scheduling and the admission control give CPU time to the high-importance thread for the periodic data collection by suspending all but one low-importance thread after the data collection is started. The aim of this experiment is to examine how quickly low-importance threads are suspended. The quickness of the thread suspension can affect the collection time of water levels.

Figure 16 shows the changes of the number of running low-importance threads. Our scheduling always suspended all but one low-importance thread just after the data collection was started every 15 s. The average suspension time was 2.2 s. The suspension time means the time from when a high-importance thread calls a scheduler until all but one low-importance thread are suspended. For the admission control, on the other hand, the number of low-importance threads was not reduced to one in several intervals, for example, from time 0 s to time 30 s. Even when all but one low-importance thread were suspended, the average suspension time was 10.2 s. This suspension time is long, compared with the interval of 15 s. Since the high-importance thread runs together with low-importance

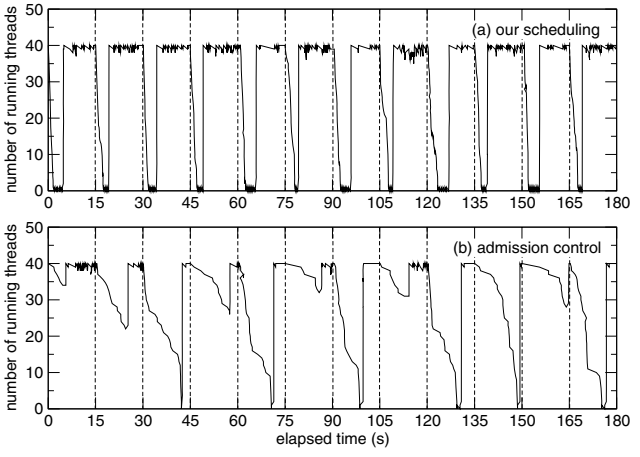


Fig. 16. The changes of the number of running low-importance threads

threads, the data collection performed by the high-importance thread tends to be delayed.

Impact of Changing Workloads. In the above experiments, JMeter sent requests to a web page showing a chart of recent changes for the last 12 h. Generating the 12 h chart was the same workload as what we used to obtain execution profile for our pointcut generator. We changed workloads so that JMeter sent requests to web pages showing charts for the last 24, 12, 6, and 3 h. As the period of a generated chart becomes smaller, the application generating a chart obtains the smaller number of water levels from the database and produces a chart in shorter time. Nevertheless, the system load becomes higher because Kasendas must process more requests per second. The aim of this experiment is to examine how well our scheduling can give sufficient CPU time to the thread for the periodic data collection under different workloads.

Figure 17 shows the average time needed for a high-importance thread to collect water levels when we changed the workloads. Our scheduling, the admission control, and Linux priority scheduling kept the average collection time to almost the same under any workloads. On the other hand, when we used the original Kasendas, the average collection time increased from 23.5 to 49.0 s at maximum and more data were lost. Kasendas tuned with Java priority scheduling was worse than the original one for the 24 and 6 h charts. Like Fig. 15, the variance of our scheduling was the smallest. That of the original Kasendas becomes larger for the shorter period. On the contrary, that of our scheduling becomes smaller as the period of the generated chart is decreasing. This indicates that threads for generating a chart of a shorter period are easier to control under application-level scheduling because the chart generation becomes relatively lighter-weight task. Table 2 shows the deadline miss ratios. In our scheduling, the deadline miss ratio was zero for every case. From these results, it is shown that our scheduling

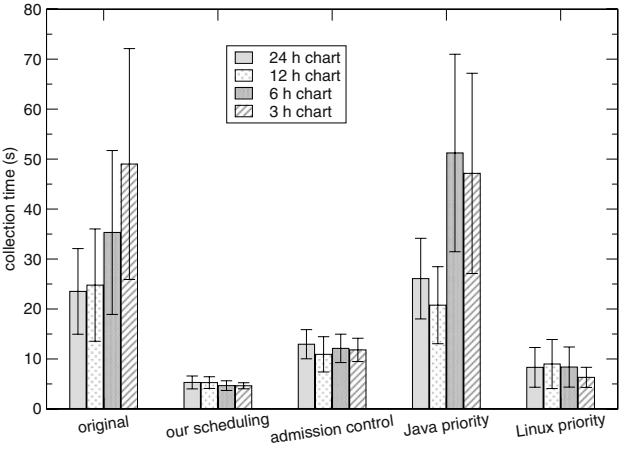


Fig. 17. The time needed for collecting water levels when the workload is changed

Table 2. The deadline miss ratios

Period	24 h	12 h	6 h	3 h
Original	0.92	0.89	0.97	0.95
Our scheduling	0.00	0.00	0.00	0.00
Admission control	0.39	0.31	0.28	0.11
Java priority	0.95	0.89	1.00	0.98
Linux priority	0.15	0.18	0.18	0.00

can control Kasendas stably even when the workload is a little different from that used in the profiled execution.

Influences to Low-Importance Threads. To examine how our scheduling affects the performance of low-importance threads, we first measured the throughput of the chart generation, which is executed by low-importance threads. Since our scheduling policy temporarily suspends low-importance threads to give sufficient CPU time to a high-importance thread, the throughput of the chart generation would be degraded. Figure 18a shows the throughput of the chart generation. Compared with the original Kasendas, the performance degradation under our scheduling was 15.7 % and larger than that under the other approaches. This is because our scheduling gave more sufficient CPU time to the high-importance thread than the other approaches. In the case of Kasendas, this level of performance degradation was acceptable because our first priority was to prevent data loss for providing reliable information.

Next, we measured the response time of a web page showing a chart. Figure 18b shows the average response time. Compared with the original Kasendas, the average response time under our scheduling increased by 18 %. The 95 % confidence intervals are (17.2, 18.2) and (19.7, 21.2) for the original Kasendas and our scheduling, respectively. Since these two do not overlap, the increase

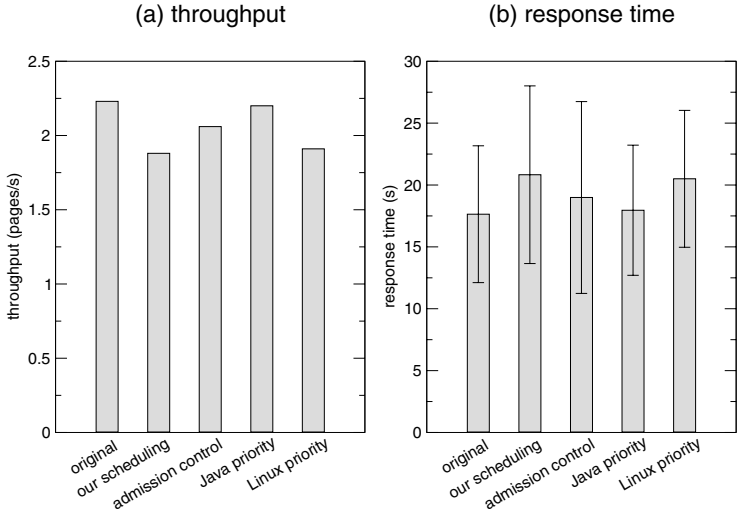


Fig. 18. The throughput of the chart generation and the response time of a web page showing a chart

is statistically significant. In addition, the variance of our scheduling was larger than that in the original Kasendas because the low-importance threads were given a low priority and all but one thread were suspended during periodic data collection.

Scheduling Overhead. To examine the scheduling overhead, we measured the throughput of the chart generation without performing the periodic collection of water levels. In our scheduling policy, low-importance threads execute the chart generation and periodically call a scheduler's method. Thus, the chart generation can cause scheduling overhead. We stopped the periodic data collection to measure pure overhead because the periodic data collection makes low-importance threads be suspended.

Compared with the original Kasendas, the throughput was not degraded in our scheduling and no scheduling overhead was measured. This is because calling a scheduler's method was very lightweight and a low-importance thread called the method at only 17 joinpoints during handling one request in our experiment. For the admission control, there was also no overhead. On the other hand, the throughputs were degraded by 3 % and 11 % with the priority scheduling by the JVM and by Linux, respectively. This is because priority scheduling gave a low priority to low-importance threads even while the periodic data collection was not performed.

5.3 Usefulness of the Pointcut Generator

We examined the impact of parameters given to our pointcut generator. The pointcut generator takes two parameters: a target interval between adjacent

joinpoints selected and the maximum occurrences of joinpoints selected by a single pointcut. In Sect. 5.2, we used the controlPoint pointcut generated with the target interval of 10 ms and the maximum occurrence of 1. For the experiments in this section, we changed the target interval to either 10 or 100 ms and the maximum occurrence to 1, 50, 100, or 200.

Generated Pointcuts. Compared with when we used a pointcut that selects all method calls without the pointcut generator, the pointcut generator dramatically reduced the number of selected joinpoints. Table 3 shows the number of generated pairs of call and withincode pointcuts and joinpoints selected by them. The number of joinpoints selected without the pointcut generator was 248661, but the number was reduced to several hundreds at most by using the pointcut generator. As the specified target interval got longer or the maximum occurrence got smaller, the number of selected joinpoints was reduced more largely. In addition, the pointcut generator generated the reasonable number of pairs of pointcuts. The number of possible pairs of pointcuts in the execution of the application generating a chart was 803 whereas the pointcut generator selected only 17 pairs of pointcuts from them at maximum.

Table 3. The numbers of generated pairs of pointcuts and joinpoints selected by them for different sets of parameters

Target interval	Maximum occurrence	Generated pointcuts	Selected joinpoints
10 ms	1	15	15
	50	16	32
	100	17	231
	200	15	309
100 ms	1	8	8
	50	9	13
	100	8	83
	200	8	83
Random		15	2034
All		803	248661

For comparison, we chose 15 pointcuts from 803 candidates at random without the pointcut generator. Such random pointcuts become the baseline for examining the goodness of the pointcuts generated by the pointcut generator. Choosing pointcuts at random only reduces the number of pointcuts whereas the pointcut generator minimizes the number of joinpoints selected in a certain period as well. When we used random pointcuts, the number of selected joinpoints was much larger than when we used the pointcut generator. This is because some pointcuts selected too many joinpoints in Kasendas. We should also compare the pointcuts generated by the pointcut generator with ideal ones, but obtaining ideal ones is very difficult for non-toy applications.

The time needed for generating these pointcuts was 20 s even when we specified 10 ms for the target interval and 200 for the maximum occurrence. The time depends mainly on the number of joinpoints included in execution profile. To examine the scalability of pointcut generation, we also measured the time needed to generate pointcuts for the web application generating a chart for the last 24 h. The number of joinpoints was 959148 in its execution profile and the time needed for pointcut generation was 103 s. Compared with the chart generation for the last 12 h, the number of joinpoints becomes 3.9 times larger while the time becomes 5.2 times longer. The increment of the time is not proportional to that of the number of joinpoints, but the time is not too long.

Influences to Scheduling Intervals. We examined how the parameters given to the pointcut generator affected the interval at which the scheduler was called at run time. Since the scheduler is called at joinpoints selected by generated pointcuts, the interval is the time between adjacent joinpoints selected. For comparison, we also examined the interval between all adjacent joinpoints and that between adjacent joinpoints selected by random pointcuts. First, we measured these intervals in single-thread execution, which was performed to obtain execution profile for pointcut generation. Only one low-importance thread ran at the same time. Figure 19 shows the averages of the observed intervals for different sets of parameters given to the pointcut generator. When we did not use the pointcut generator, the observed intervals were too small for application-level scheduling. The observed interval was 0.01 ms when all joinpoints were selected and was 0.9 ms when joinpoints were selected by random pointcuts. When we gave appropriate parameters, the pointcut generator could generate pointcuts so that the observed interval approached the target. For example, if the target interval was 10 ms and the maximum occurrence was 100, the average interval was the nearest to the target one. The observed interval tends to be smaller as the maximum occurrence became larger.

The variance of observed intervals was very large. The reason is that the pointcut generator cannot always generate pointcuts so that joinpoints selected by them occur at regular intervals. It depends on the characteristics of applications. Figure 20 plots the time when a program flow reached joinpoints selected by pointcuts. This figure shows that there were no joinpoints in parts of a program flow: time 0.0 to 0.2 s, time 0.6 to 0.7 s, and time 1.6 to 1.9 s. In the first part, the application waited for finishing database accesses. In the second part, the application created a large buffered image for a chart. In the third part, the application sent the image for the chart to the client through the network. The pointcut generator could not generate any pointcuts that selected joinpoints during these periods. By contrast, there was a part that included too many joinpoints: time 1.0 to 1.6 s, for example. In this part, JFreeChart repeated the same processing to generate a chart too many times. Even the most infrequent method call was done every 1.8 ms. This is too frequent, compared with the time quantum of 5 ms assigned to processes with the lowest priority in Linux. The pointcut generator could not generate any pointcuts so that the occurrence of

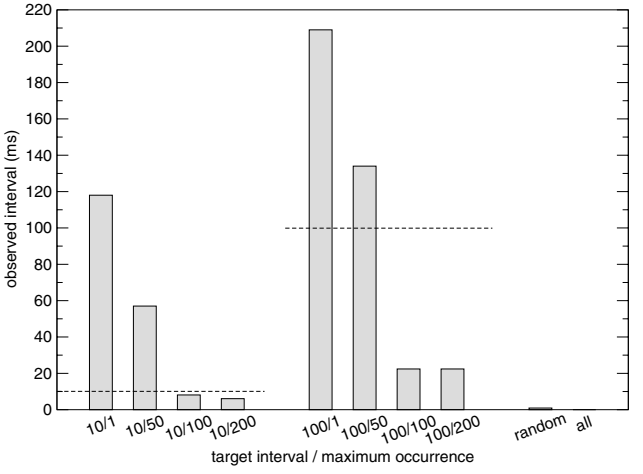


Fig. 19. The intervals at which the scheduler is called in single-thread execution

joinpoints was within the specified maximum value. Nevertheless, our scheduling worked well because it did not need to control threads too strictly.

Next, we measured the intervals in multi-thread execution. JMeter sent 10, 20, and 40 requests to a web page showing a chart in parallel. The aim of this experiment is to examine how a server load affects the observed intervals. Figure 21 shows the average of the observed intervals. At worst, each low-importance thread could call the scheduler every 1.9 s on average. For the parameters used in our experiments in Sect. 5.2, each low-importance thread called the scheduler every 1.1 s on average. This enabled stable control as shown in Sect. 5.2. This figure also shows that the observed interval was proportional to the number

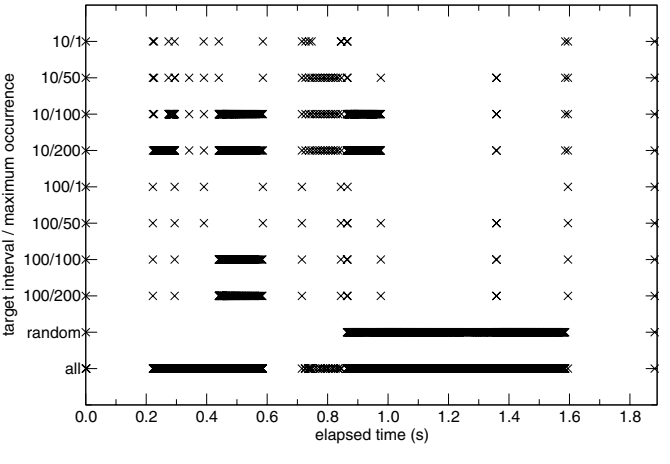


Fig. 20. The time when a program flow reaches joinpoints selected by generated pointcuts

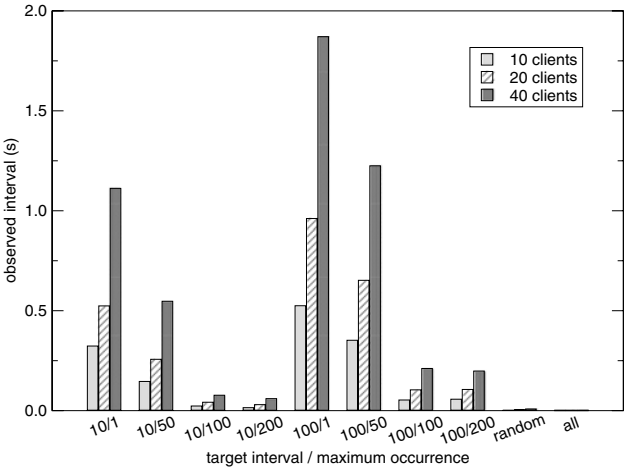


Fig. 21. The intervals at which the scheduler is called in multi-thread execution

of concurrent requests. These results show that we could predict the observed interval in multi-thread execution from that in single-thread execution.

Influences to Execution Performance. To examine how these parameters affect execution performance, we first measured the time needed for suspending low-importance threads and the time needed for periodically collecting water levels. In this experiment, JMeter sent 40 requests in parallel. The result is shown

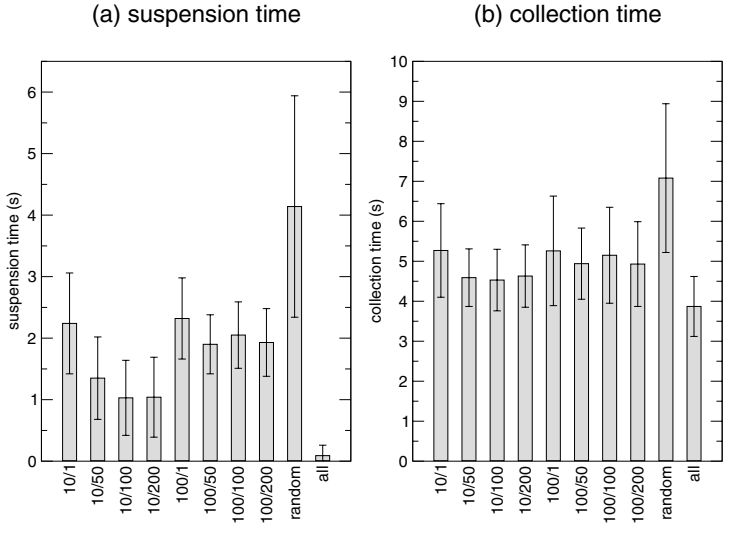


Fig. 22. The time for the thread suspension and the data collection for different sets of parameters

in Fig. 22. The time needed for thread suspension was different for each set of parameters. The time needed for the data collection reflected these differences, but it was not different as largely as the suspension time. The collection time was between 4.5 and 5.3 s and sufficient for avoiding deadline misses. On the other hand, when we used a pointcut that selected all method calls, the suspension time decreased too much and achieved short collection time. By contrast, when we used random pointcuts, the suspension time increased by a factor of two and the collection time became long. Also, its variance of the suspension time became very large because the scheduler was called at random intervals. The deadline miss ratio was zero for every case.

Next, we measured the throughput and the response time of the chart generation. The performance was almost never affected by the parameters. However, when too many joinpoints were selected due to using no pointcut generator, the performance was degraded largely. The throughput was degraded by 57 % and the response time was 2.2 times longer. This means that decreasing the number of selected joinpoints is important in terms of performance.

Finally, we examined scheduling overheads by measuring the throughput of the chart generation without the periodic data collection. When we did not use the pointcut generator, the scheduling overhead was 63 %. The pointcut generator reduced the overhead to less than 5 % for every set of parameters. For our experiments in Sect. 5.2, we experimentally selected the target interval of 10 ms and the maximum occurrence of 1 so that the scheduling overhead was minimized.

5.4 Effectiveness of Deadlock-Aware Scheduling

In this section, we used the Kasendas to which the Logging class described in Sect. 4.4 was added for introducing synchronization. When we wove the original aspect into this Kasendas and sent requests, a deadlock always occurred soon as we intended. A thread for collecting water levels blocked at the Logging.writeCollection method and it was not continued. On the other hand, when we wove the deadlock-aware version of aspect into the Kasendas, deadlocks did not occur.

Next, we measured the time needed for collecting water levels when we wove the deadlock-aware version of aspect. We also measured the times for the other four versions of Kasendas, which used their own scheduling policies. For them, we introduced synchronization by adding the Logging class. They did not cause deadlocks although we did not change their scheduling policies. The results were almost the same as Fig. 15 and the average collection time by our scheduling was the shortest (6.1 s). However, this is 0.8 s longer than that shown by the results in Fig. 15. The increment was caused by the overhead due to checking progress and the delay for detecting deadlocks.

Figure 23 shows the changes of the number of running low-importance threads. The result was a little different from that in Fig. 16a. The scheduling policy was to reduce the number of running low-importance threads to one, but this goal was often not achieved due to synchronization among low-importance threads.

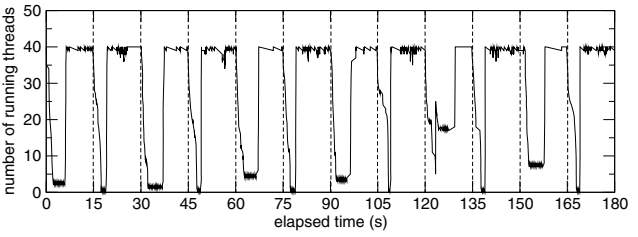


Fig. 23. The changes of the number of running low-importance threads for Kasendas with synchronization

If low-importance threads are blocked at the Logging.writeGeneration method after another low-importance thread is suspended within the same method, the blocked threads cannot yield their execution.

The performance of a chart generation under the new scheduling was similar to that in Fig. 18, but the performance degradation was larger. Compared with the original Kasendas, the throughput under the new scheduling was degraded by 20 % and the response time was 25 % longer. This is caused by the increment of the collection time. The collection time increased by 0.8 s while the response time was increased by 1.0 s.

5.5 Effectiveness of Adaptive Scheduling

First, we examined the effectiveness of our adaptive scheduling policy described in Sect. 4.4 when the periodic data collection was not performed. JMeter sent requests to both the web page showing a chart of recent changes for the last 12 h and the web page showing the up-to-date water levels. The number of concurrent requests was 40 for the former page and 100 for the latter page.

Figure 24 shows the changes of the throughput of the middle-importance task for the water-level update when we did not apply any scheduling policy. The dotted line is the observed throughput and the solid line is the average per 5 s. Even the average throughput was very unstable because the execution of the middle-importance threads was largely affected by that of the low-importance threads. The cause of the periodic changes is that the execution of the chart generation had several phases as we explained in the experiment for Linux priority scheduling in Sect. 5.2.

Figure 25 shows the changes of the throughput when we applied our adaptive scheduling policy. We set the target throughput of the middle-importance task to 150 pages/s. The average throughput per 5 s was stabilized and achieved 152 pages/s, which was very near to the target throughput. This figure also shows the changes of the maximum number of low-importance threads. Our scheduler frequently adjusted the maximum number of low-importance threads between one and six. It was almost exactly called every second according to our policy. In addition, it decreased the number just after the server started processing requests to the web page for the water-level update and increased the number just after JMeter stopped sending requests.

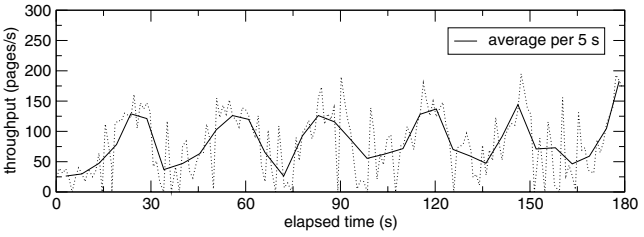


Fig. 24. The changes of the throughput of the middle-importance task under no scheduling policy

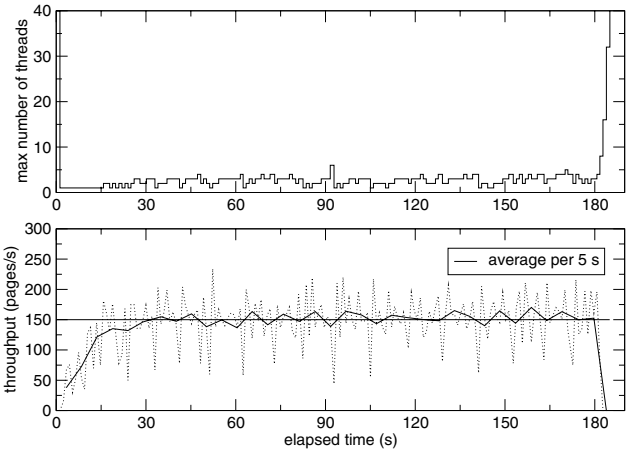


Fig. 25. The changes of the maximum number of low-importance threads and the throughput of the middle-importance task under our adaptive scheduling policy

The response time of the water-level update was also improved by a factor of two. The response time under no scheduling policy was 1.2 s while that under our scheduling policy was 0.62 s. The variance was also smaller under our scheduling policy. On the other hand, the throughput of the chart generation was degraded by adjusting the maximum number of low-importance threads. The throughput under no scheduling policy was 1.84 pages/s, but that under our scheduling policy was 1.27 pages/s. This means that the low-importance threads were given a lower priority than the middle-importance threads.

Figure 26 shows the maximum number of low-importance threads and the observed throughputs for various target throughputs. We changed the target throughput from 25 to 250 pages/s. The observed throughput was near to the target when the target was between 125 and 200 pages/s. For these target throughputs, the maximum number of low-importance threads was less than ten and the variance was small. When the target throughput was more than 225 pages/s, the observed throughput was lower than the target due to the upper limits of the system. The maximum number of low-importance threads was almost one be-

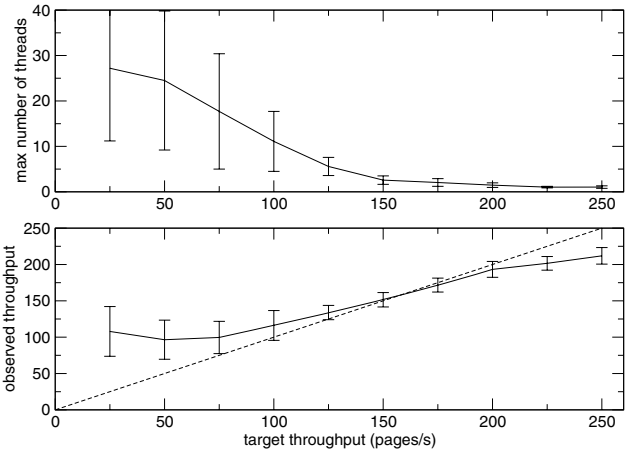


Fig. 26. The averages of the maximum numbers of low-importance threads and the observed throughputs of the middle-importance task for various target throughputs

cause it was the best strategy to minimize the impact by the low-importance threads in our policy. When the target throughput was less than 100 pages/s, the observed throughput was higher than the target one. The middle-importance threads could run too much even if the large number of low-importance threads simultaneously runs.

Next, we examined the effectiveness of our adaptive scheduling policy when the periodic data collection was performed. We set the target throughput to 150 pages/s on average when the high-importance thread for the data collection does not run. While the high-importance thread is running, we set its maxi-

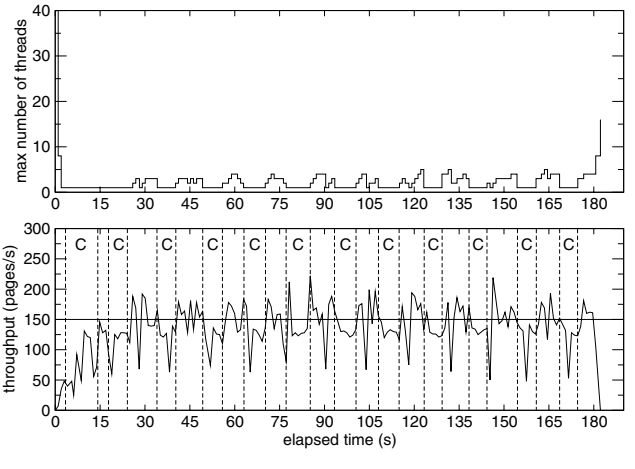


Fig. 27. The changes of the maximum number of low-importance threads and the throughput of the middle-importance task with the periodic data collection

mum throughput to 150 pages/s. To limit the maximum throughput, the maximum number of middle-importance threads was limited to 15 and each middle-importance thread waited for 100 ms before starting its execution. This means that the water-level update is performed 10 times at maximum every second for each middle-importance thread.

Figure 27 shows the changes of the maximum number of low-importance threads and the throughput of the middle-importance task. The data collection was performed during the periods marked as "C". When the high-importance thread did not run, the average throughput was 143 pages/s, which is near to the target throughput. While the high-importance thread was running, the average throughput was 122 pages/s, which is less than the specified maximum throughput. In detail, the throughput was more than 150 pages/s at the beginning of each data collection. Since the number of running middle-importance threads was decreasing to the specified one in several seconds, many middle-importance threads were running just after the data collection started. The average collection time was 6.8 s, which is 1.5 s longer than the result in Fig. 15 due to running middle-importance threads during the periodic data collection.

6 Applicability of QoSWeaver

QoSWeaver is not appropriate if the application needs accurate scheduling. The application-level scheduler by QoSWeaver slowly responds to workload changes. It may take several seconds because application threads are under the control of the underlying operating-system scheduler and the threads only voluntarily yields the allocated CPU time. For example, the result of our experiment in Fig. 16 shows that the number of the running low-importance threads was decreased from 40 to 25 in 1 s although it must be decreased to one according to the scheduling policy. The scheduling accuracy could be improved if the application program calls a scheduler more frequently. However, the scheduling overheads would be bigger. In the worst case of our experiment, in which a scheduler was called at every method call, the throughput was less than the half of the original. Thus, QoSWeaver could not be used for implementing real-time scheduling. Likewise, within a short period such as 1 s, it cannot allocate the exact CPU time computed from the priority of the thread. It can only allocate so that the average of the allocated CPU time during several seconds reflects the priority.

QoSWeaver does not work well if the application is I/O intensive and the threads frequently suspend for long time for waiting until an I/O request is completed. Since the thread must be running to call a scheduler, every I/O request should be short and infrequent. Otherwise, the scheduler would not run frequently enough to implement a specified scheduling policy. Furthermore, QoSWeaver does not work well if a small code block is repeatedly executed for long time. If a pointcut does not select a joinpoint in that code block, a scheduler will not be called for long time. On the other hand, if it selects, a scheduler will be called too frequently; the scheduling overheads will be non-negligible.

QoSWeaver assumes that the behavior of the application does not largely change for every execution. It must be similar to the behavior of the profiled execution. If the behavior of the application is categorized into several patterns, we can obtain execution profiles for each pattern, generate pointcuts for each, and merge them all. However, the accuracy of the scheduling will be degraded more as the variation of the application behavior is larger. This fact has been discussed in Sect. 3.3.

7 Concluding Remarks

In this paper, we presented QoSWeaver, which is a tool suite for developing application-level scheduling using aspects. The idea of scheduling at the application level is not new; it is a useful technique for adjusting execution performance with a minimum development cost. We showed that AOP makes this technique more realistic by separating scheduling code from applications. Furthermore, the pointcut generator provided by QoSWeaver generates appropriate pointcuts and helps developers create an application-specific scheduling mechanism, which calls a scheduler from applications periodically.

As a case study, we used a river monitoring system named Kasendas, which is a web application system initially developed by the outside corporation. Using QoSWeaver, we could successfully implement three practical scheduling policies for Kasendas. According to our experiences in the development of Kasendas, QoSWeaver made it easy to develop the scheduling policies in (1) that the scheduling policies could be developed independently of Kasendas and (2) that appropriate pointcuts were automatically generated without examining a large amount of source code of Kasendas. Through this case study, we also experimentally showed the effectiveness of our scheduling policies and the usefulness of the pointcut generator under several workloads.

One of our future work is to apply QoSWeaver to other types of applications. As discussed in Sect. 6, the application classes to which QoSWeaver is applicable are limited from the nature of application-level scheduling and profile-based pointcut generation. To analyze the applicability quantitatively, we need to examine whether QoSWeaver is applicable or not to other real applications. Another direction is to develop other scheduling policies using QoSWeaver. In this paper, we have developed three scheduling policies: proportional-share, deadlock-aware, and adaptive scheduling. To develop scheduling policies that dynamically change the frequency of calling a scheduler, depending on workload changes, it would be necessary for QoSWeaver to support dynamic weaving, for example. We would like to examine that QoSWeaver is useful in practice to achieve other classes of scheduling.

References

1. Kourai, K., Hibino, H., Chiba, S.: Aspect-oriented application-level scheduling for J2EE servers. In: Proceedings of the 6th ACM International Conference on Aspect-Oriented Software Development, pp. 1–13 (2007)

2. Hibino, H., Kourai, K., Chiba, S.: Difference of degradation schemes among operating systems. In: Proceedings of DSN 2005 Workshop on Dependable Software – Tools and Methods, pp. 172–179 (2005)
3. Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., Turnbull, M.: The real-time specification for Java. Addison-Wesley, Reading (2000)
4. Tesanovic, A., Amirijoo, M., Björk, M., Hansson, J.: Empowering configurable QoS management in real-time systems. In: Proceedings of the 4th International Conference on Aspect-oriented Software Development, pp. 39–50 (2005)
5. Duzan, G., Loyall, J., Schantz, R., Shapiro, R., Zinky, J.: Building adaptive distributed applications with middleware and aspects. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, pp. 66–73 (2004)
6. Barreto, L., Muller, G.: Bossa: A language-based approach to the design of real-time schedulers. In: Proceedings of the 10th International Conference on Real-Time Systems, pp. 19–31 (2002)
7. Åberg, R., Lawall, J., Südholt, M., Muller, G., Meur, A.L.: On the automatic evolution of an OS kernel using temporal logic and AOP. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, pp. 196–204 (2003)
8. Douceur, J., Bolosky, W.: Progress-based regulation of low-importance processes. In: Proceedings of the 17th ACM Symposium on Operating Systems Principles, pp. 247–260 (1999)
9. Newhouse, T., Pasquale, J.: A user-level framework for scheduling within service execution environments. In: Proceedings of the IEEE International Conference on Services Computing, pp. 311–318 (2004)
10. Newhouse, T., Pasquale, J.: ALPS: An application-level proportional-share scheduler. In: Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing, pp. 279–290 (2006)
11. Chang, F., Itzkovitz, A., Karamcheti, V.: User-level resource-constrained sandboxing. In: Proceedings of the 4th USENIX Windows System Symposium, pp. 25–36 (2000)
12. Elnikety, S., Nahum, E., Tracey, J., Zwaenepoel, W.: A method for transparent admission control and request scheduling in e-commerce web sites. In: Proceedings of the 13th International Conference on World Wide Web, pp. 276–286 (2004)
13. Welsh, M., Culler, D., Brewer, E.: SEDA: An architecture for well-conditioned, scalable Internet services. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles, pp. 230–243 (2001)
14. Welsh, M., Culler, D.: Adaptive overload control for busy Internet servers. In: Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (2003)
15. Sun Microsystems: JSR 220: Enterprise JavaBeans, version 3.0 (2006)
16. The Carnegie Mellon Software Engineering Institute: Capability maturity model integration, http://www.sei.cmu.edu/cmmi/
17. JBoss Group: JBoss application server, http://www.jboss.com/
18. Apache Jakarta Project: Apache Tomcat, http://tomcat.apache.org/
19. Apache Struts Project: Apache Struts, http://struts.apache.org/
20. Seasar Foundation Project: Seasar, http://www.seasar.org/
21. PostgreSQL Global Development Group: PostgreSQL, http://www.postgresql.org/

22. Object Refinery Ltd: JFreeChart, http://www.jfree.org/
23. Chiba, S., Ishikawa, R.: Aspect-oriented programming beyond dependency injection. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 121–143. Springer, Heidelberg (2005)
24. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
25. Apache Jakarta Project: Apache JMeter, http://jakarta.apache.org/jmeter/

An Exploratory Study for Identifying and Implementing Concerns in Integer Programming

Norelva Niño, Christiane Metzner, Alejandro Crema, and Eliezer Correa

Escuela de Computación, Facultad de Ciencias, Universidad Central de Venezuela,
Apartado 40388 Los Chaguaramos, Caracas 1041-A, Venezuela
norelva.nino@ciens.ucv.ve, christiane.metzner@ciens.ucv.ve,
alejandro.crema@ciens.ucv.ve, elcorrea@gmail.com

Abstract. In this paper, we analyze the suitability of refactoring the integer programming algorithms *Branch and Bound* and *Branch and Cut* with aspects implemented in Computational Infrastructure for Operations Research (COIN-OR), an open source library for Operations Research. For identifying the concerns in the code, we propose a classification of concerns in terms of requirements. We transformed the rules of an existing Aspect-Oriented Programming (AOP) refactoring catalog for *Java* to a corresponding catalog for *AspectC++* and developed a refactored version of the implemented algorithms using our transformed rules. The execution time of *Branch and Bound* and *Branch and Cut* was measured and the impact of using AOP was analyzed. The results are very encouraging and we assess that besides a customizable code, the execution time did not degrade with AOP.

Keywords: Aspect-Oriented Programming, Integer Programming, Refactoring, AspectC++.

1 Introduction

Aspect-Oriented Programming (AOP) [1] and refactorings are current techniques for coping with software evolution. The principle of *separation of concerns* in software systems is not new; Dijkstra [2] considers it as "focusing one's attention upon some aspect which does not mean ignoring the other aspects, it is just doing justice to the fact that from one aspect's point of view, the other is irrelevant." Parnas [3] considers this principle as a way of dealing with complexity guiding the decomposition or modularization of a problem. Although separation of concerns was initially used for structuring the functionalities of a program, nowadays it enables software reuse and evolution through decomposition of distinct software features, themselves overlapping in functionality as little as possible. A software feature can span more than one related functionality.

A *Concern* is any area of interest that has to be considered during the design or development of an application [4]. In Kiczales [5] AOP is presented as a new paradigm that considers and solves problems related to crosscutting concerns providing additional mechanisms to object-oriented languages for modularizing concerns affecting

A. Rashid and H. Ossher (Eds.): Transactions on AOSD V, LNCS 5490, pp. 45–71, 2009.

multiple structures in a program. Crosscutting concerns cannot be localized in only one program structure or together with other concerns. An *aspect* is the modular implementation of a crosscutting concern. If a *crosscutting concern* is implemented with traditional object-oriented programming techniques, the corresponding code will be scattered in more than one class of the application. By using AOP, the crosscutting concerns can be modularized into one aspect. *Joinpoints* are interesting points in a program's execution to be intercepted by an aspect, they can be data or execution, static or dynamic; at these points, new sections of code are executed before, after or instead of the original base code. Joinpoints are specified through a special constructor, the *Pointcut Designator*, used to quantify over programs the set of Joinpoints extending, modifying, or deleting behavior. An aspect can also define its own attributes (state) and behavior or define new attributes and methods in existing classes by using intertype declarations.

Refactoring is a fundamental technique in Software Engineering aimed at improving the quality of code while preserving its functionalities. The term, introduced by Opdyke [6], refers to a redistribution of classes, variables, and methods to facilitate future software adaptations and extensions.

The work presented here is part of an ongoing project dealing with the AOP refactoring of an open source library for Operation Research, COIN-OR. This library, which is nontrivial in size and complexity, was developed by the "Computational Infrastructure for Operations Research Project" (COIN-OR) in an initiative to spur the development of open-source software in *C++* in the Operations Research community. Most integer programming problems in the real world are NP-hard, their time complexity is nonpolynomial; therefore, when using a new technique there should be no overhead in execution time due to the technique as in this domain performance is a priority. Operation Research experts consider a secondary goal any other product quality discussed by the software engineering community; their primary goal is always performance.

COIN-OR is used in our academic context by students taking the Operations Research course where they also experience the importance of performance when solving integer programming problems. In this paper, our approach for identifying concerns is exemplified for the integer programming algorithms *Branch and Bound* (B&B) and *Branch and Cut* (B&C). Some of the concerns were implemented in *AspectC++,* the execution time and some product qualities of both versions were assessed and experimentation and outcomes are discussed. At this point, our primary goal is to determine that the average performance will not deteriorate by using AOP and our secondary goal is to obtain a code that can be customized and adapted as necessary to different strategies used by an integer programming algorithm. Our results indicate that by using AOP, on average, there is no deterioration in execution time, although solving some problems took, in the worst case, some 20% more time and that it enables a customizable and evolvable COIN-OR code. A corresponding Object-oriented (OO) refactoring using, for example, design patterns without appropriate tools would require a considerable amount of effort. The use of AOP allows code to be moved to an aspect without necessarily having identified all the sections of code were the aspect could be applied. It is to be noted that to the best of our knowledge there are no published experiences of refactorings with AOP in the Operations Research domain.

The rest of this paper is structured as follows. In Sect. 2, we describe the integer programming algorithms B&B and B&C implemented in COIN-OR and we summarize the main COIN-OR components for solving problems with these algorithms. In Sect. 3, we identify and categorize concerns in the domain of these algorithms, and the process for refactoring the code of B&B and B&C with AOP is explained and illustrated with a refactoring. The experimentation and discussion of our results are presented in Sect. 4. We finalize with our conclusions and ongoing work.

2 Context

A variety of techniques and formalisms have been defined and used for refactoring. In this paper, we are centered on what steps should be followed in a refactoring process and how to refactor. Tourwé and Mens [7] use a semiautomated approach based on logic meta programming for detecting when a design should be refactored as well as identifying which refactorings could be applied. According to these authors, three steps are identified in a refactoring process:

- *Detecting when to refactor:* the source code of an application is inspected to identify code smells [8], such as arguments in the signature of methods that are not used in the body. Code smells are the way Beck and Fowler propose to diagnose problems in existing code that could be removed by refactorings [8, chap. 3]. Code smells do not aim to provide precise criteria for when refactorings are overdue. Instead, they suggest symptoms that may be indicative of something wrong in the code. Programmers are required to develop their own sense of when a symptom indeed warrants a change. Decisions also depend on the specific aims of the programmer and the specific state and structure of the code on which he is working [9]. Inspection can be complemented with artifacts describing the architecture and design of the application, design guidelines, standards, and metrics.
- *Identifying refactorings that can be applied*: knowing which refactorings can be applied, where to use them, and having enough information to apply them. During refactoring, other refactoring possibilities may be suggested or required—known as cascaded refactorings.
- *Do the refactoring (automatically)*: implement the changes and guarantee behavior preservation. This step represents the experimentation in the refactoring process. The original definition of behavior preservation by Opdyke states that for the same set of input values, the resulting set of output values before and after refactoring should be the same [6]. An automatic refactoring tool can reduce the possibility of errors; however, we have no knowledge of an automated tool to assist the refactoring *C++* code.

Monteiro and Fernandes [9] propose a catalog of refactorings describing code transformations from *Java* to *AspectJ*-specific modularization units. The catalog consisting of 27 entries is structured into the following three groups:

- *Extraction of Crosscutting Concerns*: refactorings in this group deal with moving implementation elements related to crosscutting concerns into aspects. These refactorings are the starting point of any refactoring process. Once they have been

applied the underlying structure of the resulting aspects can be improved with refactorings of the other groups.
- *Restructuring the Internals of Aspects*: refactorings in this group deal with improving the structure (internals) of aspects such as removing duplicate code or needless complicated structures that may be a hindrance to reuse.
- *Generalization of Aspects*: refactorings in this group deal with the extraction of common code in multiple aspects, defining an aspect hierarchy suitable for reuse.

In the following section, we describe the steps of the B&B and B&C algorithms followed by a presentation of the components in COIN-OR implementing these algorithms.

2.1 The Algorithms

Through this paper, the canonical statement of the integer linear programming problem (ILP) is taken to be:

$$(P) \quad \min \quad c^t x \quad \text{subject to} \qquad x \in F(P),$$

where $F(P)$ is the set of feasible solutions of P defined as follows: $F(P) = \{x : Ax \leq b, x \geq 0, x \text{ integer}\}$

where A is a matrix and c, x, and b are vectors of appropriate dimensions. Vector x is the vector of decisions variables.

The general algorithmic framework uses three notions: separation, relaxation, and fathoming.

I. **Separation**

Problem P is said to be separated into subproblems P_1, P_2 if the following conditions hold: (i) every feasible solution of P is a feasible solution of exactly one of the subproblems and (ii) a feasible solution of any of the subproblems P_1, P_2 is a feasible solution of P.

Let $F(P_1)$ and $F(P_2)$ the feasible solutions sets of P_1 and P_2 then $F(P_1)$ and $F(P_2)$ is a partition of $F(P)$.

The subproblems P_1, P_2 are called descendants of P. Creating descendants of the descendants of P is equivalent to refining the partition of $F(P)$.

The most popular way of separating an ILP is by means of contradictory constraints on a single integer variable (the separation or branching variable). For example, P can be separated into two subproblems by means of the two mutually exclusive and exhaustive constraints $x_1 \leq 2$ and $x_1 \geq 3$.

According to Geoffrion and Marsten [10], "Our interest in separation is that it enables an obvious divide-and-conquer strategy for solving any optimization problem (P). Leaving aside for a moment the important question of *how* one separates a problem that is difficult to solve, we can sketch a rudimentary strategy of this type as follows. First make a reasonable effort to solve P. If this effort is unsuccessful, separate P into two or more subproblems, thereby initiating what will be called a candidate list of subproblems. Extract one of the subproblems from this list – call it the current *candidate problem* (CP) – and attempt to solve it. If it can be solved with a reasonable amount of effort, go back to the candidate list and extract a new candidate problem to

be attempted; otherwise, separate CP and add its descendants to the candidate list. Continue in this fashion until the candidate list is exhausted. If we refer to the best solution found so far to any candidate problem as the current *incumbent*, then the final incumbent must obviously be an optimal solution of P (if all candidate problems were infeasible, then so is P)."

II. **Relaxation**

By far the most popular type of relaxation for an ILP is to drop all integrality requirements on the variables. The SIMPLEX algorithm [11] is used to solve the resulting ordinary linear program.

The linear relaxation of P is defined as follows:

$$(P_R) \quad \min \quad c^t x \quad \text{subject to} \qquad Ax \le b, x \ge 0$$

We have now that:

(i) if P_R has no feasible solutions, then the same is true for P.
(ii) the minimal value of P is no less than the minimal value of P_R.
(iii) if an optimal solution of P_R is feasible in P, then it is an optimal solution of P.

III. **The Fathoming Criteria**

Let CP be a typical candidate problem arising from the attempt to solve P. Let CP_R be the linear relaxation of CP.

Candidate problem CP is fathomed if one of the following holds:

(i) An analysis of CP_R reveals that CP has no feasible solution,
(ii) An analysis of CP_R reveals that CP has no feasible solution better than the incumbent solution,
(iii) An analysis of CP_R reveals an optimal solution of CP, that is, an optimal solution of CP_R is found which happens to be feasible in CP.

Based on this description, the steps for exploring a branch-decision tree in B&B by using linear relaxations of the integer programming problem can be resumed as [10]:

Step 1. *Initialize Candidate List*: initialize the candidate list with an ILP and the value of the incumbent (Z*) to an arbitrarily large number.

Step 2. *Is List Empty*: stop if the candidate list is empty. Therefore, if an incumbent exists then it must be optimal for ILP, otherwise ILP has no feasible solution.

Step 3. *Select a Candidate Problem*: select one of the problems from the candidate list as the current candidate problem (CP). A LIFO strategy may be used: the problem selected is the last one added to the candidate list.

Step 4. *Solve (CP_R) the Linear Relaxation of CP*: CP_R is CP without integer constraints.

Step 5. If the outcome of Step 4 is infeasibility of CP_R, go to Step 2.

Step 6. If the outcome of Step 4 shows that CP has no better feasible solution than the incumbent, go to Step 2.

Step 7. If the outcome of Step 4 is an optimal solution of CP, go to Step 9.

Step 8. *Separate*: separate CP into subproblems CP_1 and CP_2 using the constraints $x_j \le [x_j^*]$ and $x_j \ge [x_j^*] + 1$, respectively, with x_j^* a fractional-valued x-variable in the

solution found at Step 4. The subproblems CP_1 and CP_2 are called descendants of *CP*. Add the descendants to the candidate list and go to Step 2.

Step 9. *Update Incumbent If Necessary*: a feasible solution of *CP* has been found; if the optimal value of the solution is less then Z*, record this solution as the new incumbent and set its value to Z*. Go to Step 2.

The B&C algorithm is based entirely on successively improved relaxations [10]. B&C adds new constraints or cutting planes to the relaxation in the B&B algorithm. Cutting planes are linear constraints; each new cut must crop off some feasible region of the current linear problem without also lopping off any feasible integer solutions of the original one. To use the previous algorithm of B&B, the following step for generating the cuts is introduced after Step 7:

Step 7-1. *Persist*: decide if cutting planes will be added and go to Step 4. Otherwise, go to Step 8.

The performance of these algorithms depends on the specific instance of a problem; for different problem sizes and data, the execution time can vary significantly. OR researchers strive to invent new solution methods for solving bigger, more complex models in as short a time as possible. A new solution approach is codified as a theoretical, or abstract, algorithm. To validate the approach and compare its performance against existing methods, the algorithm is implemented in software and computational studies are conducted. The implementations are usually prototypes written in a high-level language and intended for the author's use only. The results of computational studies are often provided as a complement to theorems, proofs, and algorithms in peer reviewed publications. But while the algorithmic theory is peer reviewed and openly disseminated, the software is not.

2.2 The COIN-OR Library

COIN-OR [12] is a general solver project implementing algorithms such as B&B and B&C for ILPs. The software in the COIN-OR repository is organized into individual projects, each one managed by a project manager overseeing its development. The repository is a library of interoperable software tools for building optimization tools as well as a few stand-alone packages. It is not a framework providing extension points to developers, although some of its projects such as BPS and OTS are frameworks. The projects or components used for the experimentation explained in this paper are not frameworks.

COIN-OR is written in *C++*, a frequently used language in the software industry for scientific applications. Although frequently criticized for its complexity, *C++* is the adequate language for many domains due to the runtime and memory efficiency of the generated code and a large existing code base.

In COIN-OR, the following projects or components collaborate to solve integer-programming problems with B&C [12]:

- *CLP*: **C**OIN **L**inear **P**rogramming (*COIN-OR LP*) is a *Simplex* algorithm [11] implementation.
- *CGL*: **C**ut **G**enerator **L**ibrary, a library of cutting-plane generators, provides several well-known cut generators such as: simple rounding cut generator, knapsack

cover cut generator, generalized odd hole cut generator, Gomory cut generator, lift-and-project cuts using "norm 1", probing cuts, and flow cover cut generator.

- *OSI:* **O**pen **S**olver **I**nterface, a uniform (Application Program Interface) API for calling embedded linear and mixed-integer programming solvers, defining an interface to *CLP*. Some of the operations available in *OSI* are: create a linear programming problem formulation; modify a formulation by adding rows or columns; modify a formulation by adding cutting planes computed by *CGL*; solve a formulation; extract solution information; and invoke the underlying solver's B&B component.
- *SBB*: **S**imple **B**ranch and **B**ound is a branch and cut code designed to work with any *OSI* capable solver and in particular with *CLP*. The concept of branching is based on the idea of an "object." An object (i) has a feasible region, (ii) can be evaluated for infeasibility, (iii) can be branched on, e.g., a method of generating a branching object, which defines an up branch and a down branch, and (iv) allows comparison of the effect of branching. *SBB* has been renamed as *CBC*, **C**OIN-OR **B**ranch, and **C**ut currently at version 1.01.00, due to people confusing SBB with GAMS SBB solver [13]; however, according to the documentation, this is only a renaming—the behavior has not been modified and since the version *SBB* was available and well documented at the start of this project, we continued using it.

The package diagram in Fig. 1 shows these components and their relationships.

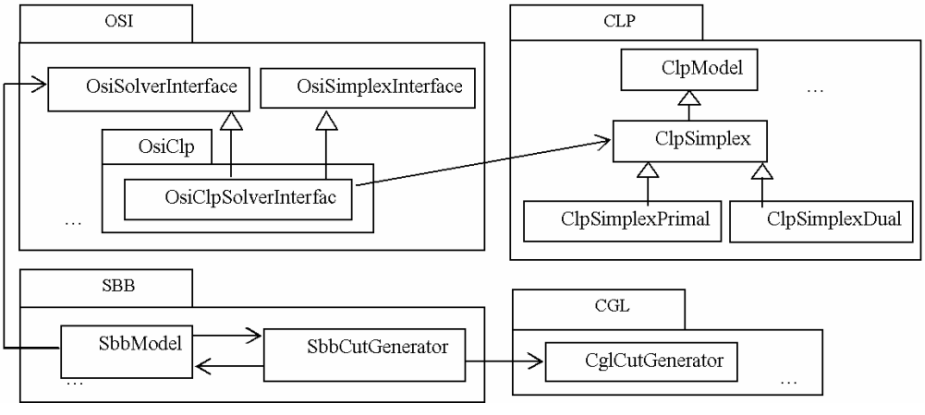


Fig. 1. A partial package diagram of components in COIN-OR

The behavior provided by the classes shown in Fig. 1 is summarized below; it is to be noted that in COIN-OR, the naming standard includes prefixing any class name with the first three letters of the component it belongs to, such as *OsiSolverInterface*.

- *ClpModel* is the base class for linear and quadratic models. This class has no knowledge of the algorithm used to solve a problem.
- *ClpSimplex* inherits from *ClpModel* and implements the *Simplex* method. Its subclasses *ClpSimplexPrimal* and *ClpSimplexDual* implement the primal and dual *Simplex* algorithms, respectively.

- *CglCutGenerator* abstract class for generating cuts.
- *OsiSolverInterface* defines an interface to a solver. Some of the central methods used when working with *OSI* are:
  - *initialSolve:* solves an initial linear programming relaxation;
  - *resolve:* solves a linear programming relaxation after a problem has been modified;
  - *branchAndBound:* invoke a solver's built-in enumeration algorithm;

Methods for handling data and for retrieving the solution are provided also by this class.

- *OsiClpSolverInterface* inheriting from *OsiSolverInterface* is the interface to getters and other methods used to solve linear problems with *CLP*.
- *SbbModel* implements a B&B algorithm and provides the following public methods:
  - *initialSolve:* finds an initial solution of the relaxed problem. Invokes an associated solver using *OsiClpSolverInterface*;
  - *branchAndBound:* implements the B&C algorithm.

A detailed description of all components and classes in COIN-OR can be found in the available documentation [12].

3 The Refactoring Process

An important element of modularity in AOP is the definition of the Joinpoints and pointcuts where aspectual adaptations will be applied. As any modularization represents requirements expressed in terms of code, we used a classification of concerns based on algorithmic requirements and a set of criteria to achieve each requirement. We distinguished the following categories and criteria:

I. **Functional Requirements of Algorithms:** behavior with probability to be modified due to software evolution or user needs is defined as a concern. Those sections of code are points of variability and they can be detected either by code analysis or by domain analysis. If code analysis is used, it is well known that the code to be changed has to be understood by whoever is doing the change. In the case of software with thousands of lines of code this task is usually complex. Ideally, the segments of code to be changed should be localized in a reduced number of modules. However, most of the time this is not the case and code not relevant to a change has to be understood too. To identify concerns of this kind, both strategies, code analysis and domain analysis, were used during the B&B and B&C refactoring.

   (a) ***Criterion: Points of Variability in B&B and B&C Algorithms.*** Behavior to be modified or customized to meet algorithmic requirements of COIN-OR users. As an example, in B&B and B&C, the number of branches for the branch-decision tree is hard coded as two branches. If a user needs to

change the number of branches, the code has to be changed. A user can customize the following strategies in B&B and B&C:

i. Choosing a subproblem from the list of pending problems.
ii. Choosing a branching variable after solving a linear relaxation.
iii. Adding new constraints for partitioning a solution space into nonintersecting subspaces in B&B.
iv. Adding cuts to a relaxed problem in B&C.

The strategies are interdependent: to partition a solution space (iii), a subproblem (i) and a branching variable (ii) have first to be selected. New constraints depend on the branching variable.

These strategies lead to the definition of the following concerns for the functional requirements of algorithms category:

1. *Choose Subproblem*: select a subproblem from the list of pending subproblems.
2. *Branching*: choose a branching variable that has taken a fractional value after solving the linear relaxation.
3. *Space Generation:* add new constraints to divide the solution space into nonintersecting subspaces.
4. *Cuts*: add constrains to a relaxed problem. Cuts are linear constraints introduced to remove a feasible region without eliminating a feasible integer solution.
5. *Halting:* constraints for stopping an execution.
6. *Pre-processing*: add new constrains to reduce a problem before using B&B or B&C.

II. **Nonfunctional Concerns:** concerns regarding quality factors of a product. In this category, our main interest is performance and usability; however, any other quality characteristics may be included.

(a) ***Criterion: Performance.*** A strategy for efficient memory use keeps in memory only those objects required at a certain point of execution and stores to disk those objects not in use. As the unused objects may be instances of any class, this concern can crosscut any class.

i. Store to disk unused objects.

(b) ***Criterion: Usability***

i. Visualization: a graphical interface for the solution tree helps users to understand the solution space of a given problem.
ii. Data input formatting: different formats are used for storing coefficients of an integer problem. Some of frequently used formats are: *Mathematical Programming System* (MPS) and *A Mathematical Programming Language* (AMPL). When an algorithm is tested, previously generated problems can be used but they may not have the adequate format. COIN-OR uses MPS for input data.

III. **Concerns Supporting a Programmer's Work.** These concerns support development or modification of a software product. When product is delivered, the lines of code introduced for this concern are removed.

(a) ***Criterion: Performance***.

i. Counters: instructions counting elements of interest for a developer.
ii. Timing: instructions recording execution time.

(b) ***Criterion: Debugging***.

i. Tracing: instructions tracing the flow of execution.

For the B&B and B&C refactoring, we could only rely on the code and the general description of the algorithms implemented in the library. We had no other software artifacts for understanding the code. The refactoring discussed in this paper is centered on category I. Depending on the results, future work eventually will deal with all concerns. We explain next how the previously described Tourwé and Mens [7] refactoring process was instantiated.

3.1 Detecting When to Refactor

To detect when a design should be refactored, Tourwé and Mens [7] indicate inspecting the source code to identify code smells looking for symptoms that may be indicative of something wrong in the code. The inspection can be complemented with artifacts, design guidelines, standards, and metrics. Due to lack of tools for identifying code smells, we inspected the source code of the components the hard way—manually, and complemented the inspection with size metrics (Table 1). A source code analyzer, *Understand for C++* [14], was used to collect measures of the program structure and traditional metrics, such as number of lines of code and comments. The manual inspection of code for identifying 14 points of variability in the four aspects analyzed in this paper consumed approximately a total of 360 person-hours. However, there are still many parts of code in the COIN-OR library to be worked on; only 4 of the 30 currently available components have been manually analyzed.

Table 1. Size metrics for selected components in COIN-OR

Components / Metrics	*SBB*	*OSI*	*CLP*	*CGL*
Classes	58	29	70	20
Files	45	51	132	45
Functions	894	619	1,811	382
Lines	25,278	33,151	94,532	28,802
Lines Blank	2,190	3,134	4,301	2,360
Lines Code (LOC)	15,161	14,960	65,262	18,912
Lines Comment	6,139	4,788	15,704	5,171
Lines Inactive	1,147	9,855	7,687	1,664
Declarative Statements	4,159	3,619	14,936	4,454
Executable Statements	8,088	8,067	37,707	11,038
Ratio Comment/Code	0.40	0.32	0.24	0.27

It is to be noted that according to the user guide and reference manual [14], adding Lines Code, Lines Comment, and Lines Blank is not equal to Lines. Some lines may contain both code and comments.

To give an idea of the size and complexity of some of the components in the library, the method *chooseBranch* of *SbbNode* in *SBB* has 412 lines of code, 191 lines of comments, a cyclomatic complexity of 71, and 999,999,999 (a possible overflow) paths, values exceeding by far the allowable thresholds; they show that this method has high risk and low stability indicating it should be refactored. According to the Software Engineering institute (SEI) [15]: "A large number of programs have been measured, and ranges of complexity have been established that help the software engineer determine a program's inherent risk and stability. The resulting calibrated measure can be used in development, maintenance, and reengineering situations to develop estimates of risk, cost, or program stability. Studies show a correlation between a program's cyclomatic complexity and its error frequency. A low cyclomatic complexity contributes to a program's understandability and indicates it is amenable to modification at lower risk than a more complex program. A module's cyclomatic complexity is also a strong indicator of its testability. A common application of cyclomatic complexity is to compare it against a set of threshold values. One such threshold set is in Table 2". These metric was useful for finding those sections of code that could be improved by refactorings.

Table 2. Cyclomatic Complexity and Risk

Cyclomatic Complexity	Risk Evaluation
1–10	A simple program, without much risk
11–20	More complex, moderate risk
21–50	Complex, high risk program
Greater than 50	Un-testable program (very high risk)

We chose *SBB*, the component responsible for implementing the branching concern, as the first component to be refactored. The strategy used for identifying relevant parts of code was rather intuitive both through modeling and detection of code smells. Name matching of methods and classes in *SBB* having explicitly the term branching in their definition were our first choice, followed by an examination of methods and objects invoked by those. This strategy does not guarantee capturing all the elements related to branching, but it provided a starting point for code examination.

The base code for class *SbbIntegerBranchingObject* and method *branch* is shown in Fig. 2 involving:

- *SbbBranchingObject:* abstract class defining the behavior for how to branch. The method *numberBranches* returns the constant value of 2 as the number of branches created for any branching object and *numberBranchesLeft* returns the number of branches left to evaluate.
- *SbbIntegerBranchingObject:* concrete subclass of *SbbBranchingObject* representing simple branching objects for an integer variable. An object can specify a two-way branch on an integer variable. For each branch, the upper and lower bounds on

the variable can be independently specified. The method *branch* performs a branch by adjusting these bounds.

The *branch* method in *SbbIntegerBranchingObject* with boolean argument normalBranch and returning a double implements the branching concern by adjusting the bounds of variable_ the branching variable and returning a change in the guessed objective on the next branch. Variables down_[0], down_[1], up_[0], and up_[1] are used for setting the bounds of a subproblem in the branch-decision tree, and according to our categorization, variable_, down_[0], down_[1], up_[0], and up_[1] correspond to points of variability in branch. The method also contains processor debugging directives.

```
class SbbIntegerBranchingObject : public SbbBranchingObject {
        …
        protected:

          double down_[2];     /// Lower [0] and upper [1] bounds for the down arm (way_ = -1)
          /// Lower [0] and upper [1] bounds for the up arm (way_ = 1)
          double up_[2];
        };

        double SbbIntegerBranchingObject::branch(bool normalBranch) {
        if (model_->messageHandler()->logLevel()>2&&normalBranch)
        print(normalBranch);
        numberBranchesLeft_--;
        int iColumn = model_->integerVariable()[ variable_ ];
        if (way_<0)    {
        #ifdef SBB_DEBUG
        { double olb,oub ;
        olb = model_->solver()->getColLower()[iColumn] ;                    Debugging instructions
        oub = model_->solver()->getColUpper()[iColumn] ;
        printf("branching down on var %d: [%g,%g] =>[%g,%g]\n", iColumn,olb,oub,down_[0],down_[1]);}
        #endif
        model_->solver()->setColLower( iColumn, down_[0]);
        model_->solver()->setColUpper( iColumn, down_[1] );
        way_=1;
        }  else    {

        #ifdef SBB_DEBUG
        {  double olb,oub ;
        olb = model_->solver()->getColLower()[iColumn] ;                    Debugging instructions
        oub = model_->solver()->getColUpper()[iColumn] ;
        printf("branching up on var %d: [%g,%g] => [%g,%g]\n", iColumn,olb,oub,up_[0],up_[1]); }
        #endif
        model_->solver()->setColLower(iColumn, up_[0] );
        model_->solver()->setColUpper(iColumn, up_[1]  );
        way_=-1;
        }
        return 0.0;
        }
```

Fig. 2. Code of *branch* method in class *SbbIntegerBranchingObject*

3.2 Identifying Refactorings That Can Be Applied

Identification of which refactoring to apply can be highly dependent on the application domain. Any appropriate catalog of refactoring can be useful for identifying refactoring possibilities as it is an indication of where to look. Considering that the language extensions for *Java* and *C++* are grounded on the same AOP concepts and

their definition and implementation are equivalent, as shown in Appendix A, Table 12, we adapted the refactoring rules in the catalog proposed by Monteiro and Fernandes [9] to *AspectC++*. The correspondence of each rule in this catalog to the *AspectC++* counterpart is summarized in Tables 3, 4, and 5 below.

Table 3. Refactoring for *Extraction of Crosscutting Concerns*

#	Rule Name	Recommended action (*AspectJ*)	Applies to *AspectC++*?	Recommended action for *AspectC++*
1	Change Abstract Class to Interface	Turn the abstract class into an interface and adjust subclasses.	N	*C++* has multiple inheritance.
2	Extract Feature into Aspect	Extract all the implementation elements related to a feature into an aspect.	Y	Same as for *AspectJ*.
3	Extract Fragment into Advice	Create a *pointcut* capturing the *Joinpoint* and context. Extract the code fragment to an *advice* based on the *pointcut.*	Y	Same as for *AspectJ*: *pointcut*, *Joinpoint* and *advice* are also features in *AspectC++*
4	Extract Inner Class to Stand-alone	Eliminate dependencies from the enclosing class and turn the inner class into a stand-alone class.	Y	Same as for *AspectJ*. The inner class turns into a stand-alone class.
5	Inline Class within Aspect	Move the class within the aspect.	Y	Same as for *AspectJ*.
6	Inline Interface within Aspect	Move the interfaces inside the aspect.	N	Rule 6 equivalent to Rule 5 in *C++*.
7	Move Field from Class to Intertype	Move the field to the aspect as an *intertype* declaration.	Y	Move the field to aspect by using an advice for static Joinpoints (*introduction*).
8	Move Method from Class to Intertype	Move method into aspect encapsulating the secondary concern as an *intertype* declaration.	Y	Move method to aspect by using an advice for static Joinpoints (*introduction*).
9	Replace Implements with Declare Parents	Replace *implements* in the class with a *declare parents* in the aspect.	Y	*Introduction* of a base class using an advice for static Joinpoints with base class in the aspect.
10	Split Abstract Class into Aspect and Interface	Move all concrete members from the abstract class to an aspect and turn class into interface.	N	*C++* has multiple inheritance.

For the refactoring of the *branch* method, the following refactoring rules defined in the catalog and belonging to the category "*Extraction of Crosscutting Concerns*" were applied:

- Rule # 2: "*Extract Feature into Aspect*": the rule should be used when the code related to a feature is scattered across several methods and classes, tangled with unrelated code; the recommended action is to extract all the implementation elements related to the feature into an aspect.
- Rule # 3: "*Extract Fragment into Advice*": the rule should be applied when part of a method is related to a concern whose code is being moved to an aspect; the recommended action is to create a Pointcut capturing the required Joinpoint and move the code fragment to an appropriate advice based on the Pointcut.

Table 4. Refactoring for *Restructuring the Internals of Aspects*

#	Rule Name	Recommended action (*AspectJ*)	Applies to *AspectC++*?	Recommended action for *AspectC++*
1	Extend *Marker*[1] Interface with Signature	Add an *intertype* abstract declaration of the specific signature to the interface.	Y	Add an *introduction* for an abstract declaration of the specific signature.
2	Generalize *Target* Type with *Marker* Interface	Replace references to specific types with a marker interface and make the specific types implement the marker interface.	Y	Replace references to specific types with an abstract declaration implemented by the specific types.
3	Introduce Aspect Protection	Declare the *intertype* member as public and place a declare *error* preventing its use outside the aspect inheritance chain.	Y	Declare the member introduction as public and use *advice around* without *proceed()*, indicating by *!within(…)* that the Joinpoint is not in the scope of the inheritance chain. In the advice body invoke a new aspect method displaying a warning message.[2]
4	Replace *Intertype* Field with Aspect Map	Replace *intertype* declarations with a structure owned by the aspect mapping the additional state and *target* objects.	Y	Replace *introduction* declarations with a structure owned by the aspect mapping the additional state and *target* objects.
5	Replace *Intertype* Method with Aspect Method	Replace the *intertype* method with an aspect method getting the *target* object as parameter.	Y	Replace method *introduction* with an aspect method getting the *target* object as parameter.
6	Tidy Up Internal Aspect Structure	Tidy up the internal structure of the aspect by removing duplication and dependencies on case specific *target* types.	Y	Tidy up the internal structure of the aspect by removing duplication and dependencies on case specific *target* types.

These two refactorings were applied to methods and classes of the *SBB* component to implement the concerns 1–4 identified in category I, using criterion a. Coefficients in the ILP (e.g., *c*, *A*, *b*) represent the input data for executing COIN-OR and, therefore, refactorings are not applied to an ILP. As described before, the manual inspection of the code for identifying 14 points of variability in the four aspects analyzed in this paper required a total of approximately 360 person–hours.

The authors of the catalog, when applying "*Extract Feature into Aspect,*" directly add attributes with no getter/setter methods to the code of a class. Our solution is the definition of a new refactoring rule belonging to the group "*Restructuring the Internals of Aspects*." This rule will not alter the base code and is defined as:

– "*Incorporate Getters/Setters using Intertype*": situation arising when the aspect code accesses the value of an attribute and the class does not provide methods for getting/setting those values. The base code is not to be changed; the recommended action is defining an intertype declaration for getter/setter methods in the aspect.

[1] *Marker* interface also called "tag" interface since they tag all derived classes into categories based on their purpose. It does not actually define any fields. It is used to "mark" *Java* classes supporting a certain capability—the class marks itself as implementing the interface. For example, the *java.lang.Cloneable*, *java.io.Serializable.*

[2] In http://www.aspectc.org/pipermail/aspectc-user/2006-January/000874.html a different translation of "declare error" results in a compile-time error in *AspectC++*. Such a translation can be implemented with a C++ template.

Table 5. Refactorings for *Generalization of Aspects*

#	Rule Name	Recommended action (*AspectJ*)	Applies to *AspectC++*?	Recommended action for *AspectC++*
1	Extract Superaspect	Move the common features to a super aspect.	Y	Move common features to a super aspect.
2	Pull Up *Advice*	Move the *advice* to the super aspect.	Y	Move *advice* to the super aspect.
3	Pull Up *Declare Parents*	Move the *declare parents* to the super aspect.	Y	Move the generic *advice* with *base class* to the super aspect.
4	Pull Up *Intertype Declaration*	Move the *intertype* declaration to the super aspect.	Y	Move member introduction to the super aspect.
5	Pull Up *Marker* Interface	Move the marker interfaces to the super aspect.	Y	Move abstract class to the super aspect.
6	Pull Up *Pointcut*	Move the *pointcut* to the super aspect.	Y	Move the *pointcut* to the super aspect.
7	Push Down *Advice*	Move the *advice* to the subaspects that use it.	Y	Move *advice* to the subaspects using it.
8	Push Down *Declare Parents*	Move the *declare parents* to the subaspects where it is relevant.	Y	Move generic *advice* with *base class* to the subaspects where it is relevant.
9	Push Down *Intertype Declaration*	Move the *intertype* declaration to the sub aspect where it is relevant.	Y	Move member introduction to the sub aspect where it is relevant.
10	Push Down *Marker* Interface	Move the marker interface to those subaspects.	Y	Move abstract class to the sub aspect.
11	Push Down *Pointcut*	Move the *pointcut* to those subaspects that use it.	Y	Move the *pointcut* to those subaspects that use it.

3.3 Do the Refactoring

The "*Extract Feature into Aspect*" rule moves the code for adjusting bounds to the *branching* aspect. The rule "*Extract Fragment into Advice*" was used to create a Joinpoint intercepting the execution of the method *branch*; now the aspect can set different strategies for the branching variable, depending on user's requirement. Direct access to class attributes variable_, down_[0], down_[1], up_[0], and up_[1] was transformed. The advice before allows changing and accessing those bounds through setter / getter methods. Reference to an object was retrieved through *that(type pattern)*, and for intercepted methods with arguments, the *args(type pattern, …)* was used.

For the refactoring of the code in Fig. 2, we used **that**(*TypePattern*) for filtering objects depending on the current object type and **args**(*TypePattern)* for filtering objects depending on the arguments types of a dynamic Joinpoint. Instead of *TypePattern* it is also possible to pass the name of a context variable to which context information from the Joinpoint will be bound.

From the syntactical perspective, an *aspect* in *AspectC++* is very much like a class in *C++*. However, besides member functions and data elements, an aspect can additionally define *advice*. Advice defines how an aspect affects the program at a given pointcut. Advice for dynamic Joinpoints is used to alter the flow of control when the Joinpoint is reached. The following kinds of advice are supported for implementing additional behavior: before, after, and around [16]. Advice for dynamic Joinpoints is defined with the syntax: **advice** <*target-pointcut*> : (**before | after| around**) (<*arguments*>) { <*advice-body*> }

The before and after advice bodies are executed before or after the event described by <*target-pointcut*>. The around advice body is executed instead of the event.

Pointcut functions can be used to filter or select Joinpoints with specific properties. Some of them can be evaluated at compile time while others yield conditions that have to be checked at run time.

After refactoring, the code is basically the same as before refactoring in Fig. 3; it still contains the responsibility of the method and debugging instructions.

```
class SbbIntegerBranchingObject : public SbbBranchingObject {
   protected:    double down_[2];     /// Lower [0] and upper [1] bounds for the down arm (way_ = -1)
                 double up_[2];       /// Lower [0] and upper [1] bounds for the up arm (way_ = 1)
   …};
      double SbbIntegerBranchingObject::branch(bool normalBranch) {
      if (model_->messageHandler()->logLevel()>2&&normalBranch)
      print(normalBranch);
      numberBranchesLeft_--;
      int iColumn = model_->integerVariable()[ variable_ ];
      double down0 = down_[0];
      double down1 = down_[1];
      double up0 = up_[0];
      double up1 = up_[1];
      if (way_<0)   {
      #ifdef SBB_DEBUG
      { double olb,oub ;
      olb = model_->solver()->getColLower()[iColumn] ;                    Debugging instructions
      oub = model_->solver()->getColUpper()[iColumn] ;
      printf("branching down on var %d: [%g,%g] =>[%g,%g]\n", iColumn,olb,oub,down_[0],down_[1]);}
      #endif
      model_->solver()->setColLower( iColumn, down0);
      model_->solver()->setColUpper( iColumn, down1);
      way_=1;
      } else   {
      #ifdef SBB_DEBUG
      {  double olb,oub ;
      olb = model_->solver()->getColLower()[iColumn] ;                    Debugging instructions
      oub = model_->solver()->getColUpper()[iColumn] ;
      printf("branching up on var %d: [%g,%g] => [%g,%g]\n", iColumn,olb,oub,up_[0],up_[1]); }
      #endif
      model_->solver()->setColLower(iColumn, up0);
      model_->solver()->setColUpper(iColumn, up1);
      way_=-1;
      }
      return 0.0;
      }
```

Fig. 3. After refactoring *branch* method in *SbbIntegerBranchingObject*

However, now branch can be customized with different strategies without modifying the source code by setting variable_ and bounds down_[0], down_[1], up_[0], and up_[1] making the code easier to change when a bound strategy needs to be changed. Also, six methods were added to the class *SbbIntegerBranchingObject* using "*Incorporate Getters/Setters using Intertype*." The points labeled "A1" in the *branching* aspect in Fig. 4 show the code of the advice. Regarding the debugging instructions used in the base code as the one's shown in Fig. 2, they are defined using processor directives; a quite obvious refactoring possibility is to have an exception handling aspect dealing with all the exceptions. However, in the code of B&B and B&C and

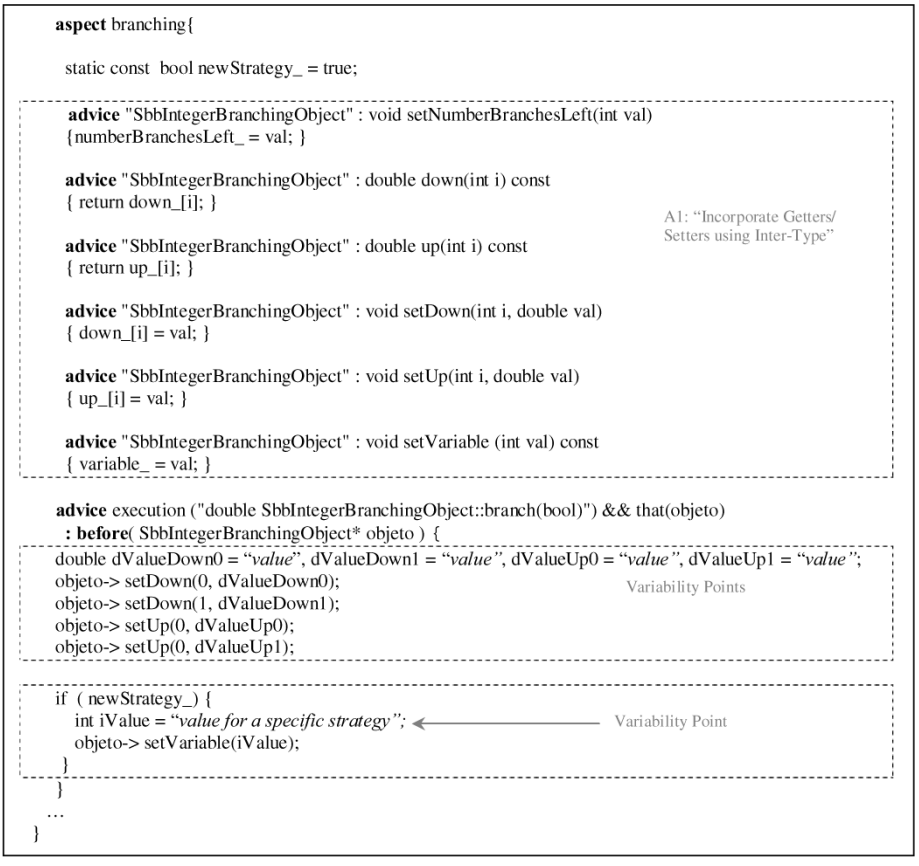


Fig. 4. *Branching* Aspect

generally in COIN-OR, there are too many of them scattered in the code to be manually removed with a reasonable amount of effort; therefore, they were not refactored at this point in time. The refactored code is available at <http://baobab.ciens.ucv.ve/refactored_code/>

The refactored and woven *C++* code was evaluated using *UnderstandC++* and the resulting size measures are shown in Table 6. Each *.cpp file has been woven with the aspect code and the *AspectC++* code transformed to *C++* has been inserted in all methods. The size of the measured items—exception made of the number of files—has increased considerably; on the other hand, inspection of cyclomatic complexity of classes and methods intercepted by the branching aspect (Table 7) shows a reduction in this measure and, therefore, a reduction of risk in those classes. The cyclomatic complexity is reduced as a consequence of moving those parts of the code related to a strategy selection to an aspect. These results confirm once more the inadequacy of using only size metrics for code assessment: it does not reflect a cleaner code and the flexibility gained by allowing changing the strategies in the algorithms.

Table 6. Size metrics for *C++* and the woven code

Metrics	COIN-OR without AOP	COIN-OR refactored with Branching Aspect
Classes	156	2,829
Files	194	194
Functions	3,430	34,309
Lines	141,899	797,648
Lines Blank	8,415	81,165
Lines Code	93,194	328,703
Lines Comment	29,112	283,828
Lines Inactive	8,419	67,017
Declarative Statements	21,537	136,216
Executable Statements	51,514	115,798
Ratio Comment/Code	0.31	0.86

Table 7. Cyclomatic complexity for *C++* and woven *SBB* code

Class	Coin-Or without AOP		Coin-Or with branching aspect			Coin-Or without AOP	Coin-Or With branching aspect
	Cyclomatic complexity				Method	Cyclomatic complexity	
	Avg.	Max	Avg.	Max			
SbbBranchDecision	2	5	1	1	bestBranch	5	1
SbbBranchDefaultDecision	4	22	1	1	betterBranch	22	1
SbbBranchingObject	1	2	1	2	numberBranches	1	1
					numberBranchesLeft	1	1
SbbIntegerBranchingObject	1	3	1	3	branch	3	1
SbbSimpleInteger	1	4	1	4	createBranch	3	1
SbbNode	4	71	4	71	branch	1	1
					numberBranches	2	1
SbbNodeInfo	2	10	2	10	numberBranchesLeft	1	1
					branchedOn	1	1

4 Evaluation and Discussion

B&B and B&C without aspects and with aspects are evaluated in this section. As the computational effort for executing B&B and B&C are highly dependent on the data, we generated and selected input data for the trials with an execution time of at most 13 h.

In this section, we describe in detail how the data for experimentation was generated and how the trials for measurement were setup. All trials were executed in *AspectC++/C++* on an *Intel Pentium 4* processor, 3.0 GHz, 1 GB RAM, with *Microsoft Windows 2000, Visual Studio .NET 2003 v7.0,* and *AspectC++ v1.0.0.2*. The same compiler was used for execution without and with aspects; the static *AspectC++* weaver *ac++* is not a target platform compiler, but a source-to-source weaver that transforms *AspectC++* code into *C++* code. This means, *ac++* cannot be called to

build an executable, but to build a woven version of the source code. This woven version has to be compiled using any standard-compliant *C++* compiler as back end, in our case the *Visual Studio* compiler.

For the trials, 156 classes were used to compile and run problem instances: 34 classes belonging to the *SBB* component (7,737 LOC), 14 classes from *OSI*, (6,395 LOC), 32 classes from *CLP* (40,429 LOC), 3 classes from *CGL* (5,568 LOC), and 73 classes from *COIN* (27,627 LOC).

4.1 Experimental Data Generation

General solvers, as COIN-OR, are mainly used for solving problems with unknown specific structure; however, we chose the *Multiconstrained Knapsack Problems* (MKP) and *General Assignment Problems* (GAP) to use an appropriate data generation procedure. MKP and GAP problems are the input data for COIN-OR without and with aspects.

MKP is an NP-hard problem used frequently by the industry and the operations research community for formulating problems dealing with resource assignment [11]. Improving existing algorithms for solving MKP is an active research topic. MKP can be stated as the following general problem: given m quantifiable independent properties (e.g., weight, volume) with each property having a capacity b_i, n items, and a knapsack, we want a solution for filling the knapsack with items maximizing the total benefit without exceeding the capacity restrictions. Every item has an associated benefit c_j, and a_{ij} is the contribution of item j to the property i. Selection of item j is represented by $x_j = 1$; if it is not selected then $x_j = 0$. This is formulated mathematically as:

$$\max \quad \sum_{j=1}^{n} c_j x_j \qquad \textit{subject to} \qquad \sum_{j=1}^{n} a_{ij} x_j \le b_i, \quad i = 1,...,m$$

$$x_j \in \{0,1\} \quad j = 1,...,n \quad \textit{where} \quad b_i \ge 0, \;\; c_j \ge 0, \;\; a_{ij} \ge 0, \;\; i = 1,...,m, \quad j = 1,...,n.$$

In this experimentation, when building random problem instances, the coefficient values c_j and a_{ij} were realizations of *Uniform* random variables generated according to the equations [17] below and obtained using Carnahan's mathematical library [18].

$$c_j \sim U(c_{\min}, \; c_{\max}) \quad \textit{with} \quad c_{\min} = 1, \; c_{\max} = 1000, \quad j = 1,...,n$$

$$a_{ij} \sim U(a_{\min}, \; a_{\max}) \quad \textit{with} \quad a_{\min} = 1, \; a_{\max} = 1000, \; i = 1,...,m, \; j = 1,...,n$$

$$b_i = \alpha \;\; x \;\; \sum_{j=1}^{n} a_{ij}, \quad i = 1,...,m \quad \textit{where} \quad \alpha = 0.5.$$

The values for all the coefficients in each problem were stored in a *Mathematical Programming System* (MPS) file; each file represents one MKP problem instance with m restrictions (quantifiable properties) and n variables (items). The test data sets used were of size: (20x150), (20x200), (20x250), (20x300), (25x150), (25x200), (25x250), and (25x300); four different problems were generated for each size and identified in the tables found in the next section as problem number i, with $i = 1, 2, 3, 4$.

In GAP, an optimal assignment of m agents to n tasks is to be found such that every task will be performed by exactly one agent. Every agent i requires a certain quantity r_{ij} of a resource to perform task j, subject to the resource availability b_i that an agent i may use. The mathematical formulation for this problem is:

$$\max \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} \quad \textit{subject to} \quad \sum_{j=1}^{n} r_{ij}x_{ij} \le b_i, \quad i = 1,...,m$$

$$\sum_{i=1}^{m} x_{ij} = 1, \quad j = 1,...,n, \qquad x_{ij} \in \{0,1\}, \quad i = 1,...,m, \quad j = 1,...,n$$

$$\textit{where} \quad b_i \ge 0, \; c_{ij} \ge 0, \; r_{ij} \ge 0, \; i = 1,...,m, \quad j = 1,...,n.$$

with $n' = m \; x \; n$, the number of variables and $m' = m + n$ the number of restrictions. (1)

In the generation of problem instances, the parameters are calculated as [17]:

$$c_{ij} \sim U(c_{\min}, \; c_{\max}) \quad \textit{with} \quad c_{\min} = 1 \quad c_{\max} = 1000, \quad i = 1,...,m, \quad j = 1,...,n$$

$$r_{ij} \sim U(a_{\min}, \; a_{\max}) \quad \textit{with} \quad a_{\min} = 1 \quad a_{\max} = 50, \; i = 1,...,m, \; j = 1,...,n.$$

$$b_i = \frac{\sum_{j=1}^{n} r_{ij}}{m \quad x \quad \alpha}, \qquad i = 1,...,m \quad \textit{where} \quad \alpha = 0.85.$$

Each MPS file represents an instance of a GAP problem with m agents and n tasks. The test datasets were of size: (30x70), (30x80), (30x90), and (30x100) and four different problems were generated for each size.

Regarding the availability of data in MPS format the size of the problems could not be controlled and in some cases, the problem structure could not be identified or the amount of resources required exceeded our computing capacity. The data we generated and the problem structure used for experimentation is available at <http://baobab.ciens.ucv.ve/mps_files/>

4.2 Experimental Results

For each problem, five runs were executed without AOP and with AOP using the same algorithms/settings. In Tables 8, 9, and 10, the mean execution time of the five runs for each problem are shown. It is not relevant to show the time for one execution or for five executions. Experimentally, each run has an estimate of the execution time for solving an integer problem. The value can be different for different runs; statistically at least five independent runs for each problem are needed to have a valid result of mean execution time. All problems were solved (an optimal solution was found) and their mean execution time (in seconds) was calculated. In the tables of this section, the following entries are shown: in columns 3, 4, 5, 6, and 7, respectively, the mean execution time without AOP and with AOP. Improvement was measured by the Aspect Performance Improvement (API) and defined as the mean time difference; a positive value indicates that mean execution time with AOP is less than mean execution time without AOP, and a negative value indicates the opposite. API% in

columns 8, 9, 10, and 11 is the percentage variation in mean execution time with respect to the mean execution time without aspects.

4.2.1 MKP Problems

For MKP problems with m = 20 restrictions and varying n, the number of variables, results are shown in Table 8.

Table 8. Results for MKP problems with m = 20

n	Pr	Mean CPU time (s)					Difference API%			
		Without AOP	With AOP							
			Branching	Space generation	Choose Subproblem	Cuts	Branching	Space generation	Choose Subproblem	Cuts
150	1	2,322.24	2,141.25	2,267.11	2,182.60	2,138.98	7.79	2.37	6.01	7.89
150	2	487.72	437.34	513.68	465.00	488.14	10.33	-5.32	4.66	-0.09
150	3	48.63	45.11	48.09	47.67	45.69	7.24	1.11	1.97	6.05
150	4	1,033.56	865.70	1,044.19	1,035.01	878.42	16.24	-1.03	-0.14	15.01
200	1	869.11	747.31	757.07	749.12	735.07	14.01	12.89	13.81	15.42
200	2	314.02	274.95	291.19	279.32	283.28	12.44	7.27	11.05	9.79
200	3	721.73	653.33	669.64	650.71	620.10	9.48	7.22	9.84	14.08
200	4	68.16	57.83	62.22	61.61	58.30	15.16	8.71	9.61	14.47
250	1	4,195.28	3,655.20	3,770.62	3,684.57	3,764.39	12.87	10.12	12.17	10.27
250	2	1,827.56	1,598.25	1,675.02	1,673.28	1,657.31	12.55	8.35	8.44	9.32
250	3	7,632.06	6,678.72	6,636.12	6,610.51	6,478.16	12.49	13.05	13.38	15.12
250	4	1,682.57	1,438.09	1,513.94	1,503.84	1,421.74	14.53	10.02	10.62	15.50
300	1	1,961.78	1,594.25	1,636.56	1,619.07	1,652.52	18.73	16.58	17.47	15.76
300	2	17,146.48	15,172.8	15,910.56	15,866.80	14,480.16	11.51	7.21	7.46	15.55
300	3	39,055.34	32,258.7	33,588.00	32,794.46	32,329.42	17.40	14.00	16.03	17.22
300	4	4,486.00	4,197.45	4,238.21	3,913.18	3,760.42	6.43	5.52	12.77	16.17

The mean execution time with AOP in MKP problems of size (20xn) varying n from 150 to 300 in steps of 50 is less than the execution time without aspects. The best performance is obtained with *Branching* being this also the aspect with a pointcut *execution(…)* and the greatest number of advice around providing ten points of variability. Aspects containing advice with *cflow(...)*, as in *Choose Subproblem* perform worse with a higher mean execution time than those using other kinds of advice such as *Cuts*. This outcome for dynamic pointcuts confirms previously published results by Lohmann et al. [19] for the domain of operating systems. The values for API % increase as n increases, as can be seen when n = 250, 300 where API% is best at 14.53% and 18.73%, respectively. Using aspects is therefore recommended.

In Table 9, the results for MKP problems with m = 25 restrictions and varying n, the number of variables, are shown. Mean execution time with AOP in MKP problems of size (25xn) with n varying from 150 to 300 in steps of 50 is less than the mean execution time without aspects. As n increases API% increases, when n = 200, 250, 300 the best improvement is about 18% and 22% with *Branching* and *Cuts*, respectively.

Table 9. Results for MKP problems with $m = 25$

n	Pr		Mean CPU time (s)				Difference API%			
		Without AOP	With AOP							
			Branching	Space generation	Choose subproblem	Cuts	Branching	Space generation	Choose subproblem	Cuts
150	1	11.20	10.37	10.66	10.56	10.77	7.41	4.82	5.71	3.84
150	2	128.68	129.86	130.54	121.09	125.28	-0.92	-1.45	5.90	2.64
150	3	171.80	160.69	170.47	172.03	168.17	6.47	0.77	-0.13	2.11
150	4	238.81	202.94	199.21	197.53	198.59	15.02	16.58	17.29	16.84
200	1	547.83	483.38	436.45	432.77	432.59	11.77	20.33	21.00	21.04
200	2	19,552.20	16,765.4	17,181.90	17,007.54	16,668.38	14.25	12.12	13.01	14.75
200	3	2,160.03	1,807.97	1,885.04	1,855.75	1,863.88	16.30	12.73	14.09	13.71
200	4	1,530.76	1,258.30	1,256.39	1,246.39	1,250.85	17.80	17.92	18.58	18.29
250	1	6,290.18	5,434.00	5,563.24	5,485.02	5,476.52	13.61	11.56	12.80	12.94
250	2	20,608.66	17,275.20	17,130.90	16,771.54	16,559.12	16.18	16.88	18.62	19.65
250	3	3,773.57	3,083.01	3,160.52	3,131.52	3,051.52	18.30	16.25	17.01	19.13
250	4	6,724.44	5,600.17	5,801.97	5,753.83	5,689.53	16.72	13.72	14.43	15.39
300	1	36,004.36	29,699.90	30,112.36	29,674.20	28,125.64	17.51	16.36	17.58	21.88
300	2	9,105.36	7,420.92	7,487.58	7,456.95	7,319.84	18.50	17.77	18.10	19.61
300	3	6,215.15	5,115.19	5,214.91	5,164.90	5,221.83	17.70	16.09	16.90	15.98
300	4	6,573.91	5,469.09	5,570.67	5,451.62	5,335.78	16.81	15.26	17.07	18.83

When *m*, the number of restrictions, increases API% also increases. In Fig. 5, the relation between mean API% and size is shown for the *Branching* aspect. According to these values, the use of aspects is recommended.

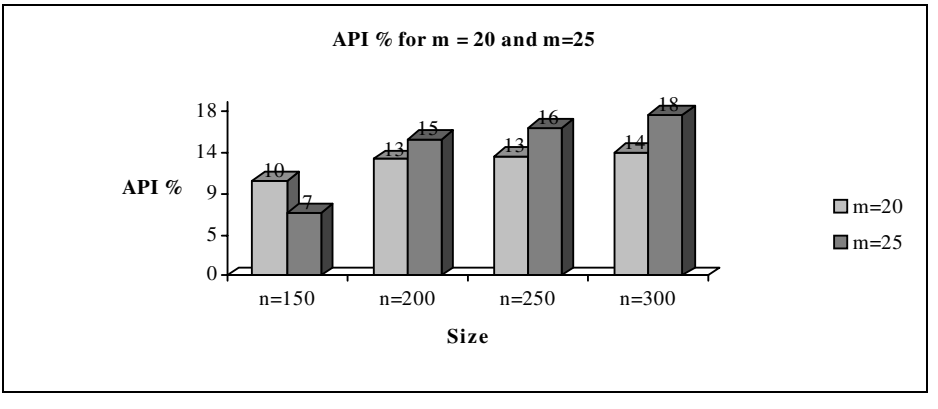


Fig. 5. Size vs. Improvement with aspect *Branching*

4.2.2 GAP Problems

The results for GAP problems with $m = 30$ agents and varying n, the number of tasks, from 70 to 100 in steps of 10 are shown in Table 10; according to Eq. 1 the values for the number of variables $n' = 2100, 2400, 2700, 3000$ and the number of restrictions $m' = 100, 110, 120, 130$ can be derived.

As can be seen from Table 10, the API% for *Branching* is between -7.06% and 7.40%; for *Space Generation*, API% is between -5.14% and 8.03%; for *Choose Sub Problem* API% is between -4.43% and 8.62%; and for *Cuts*, API% is between -4.56% and 6.93%; this indicates that there is practically no gain (or loss) in using aspects. However, it is to be noted that the execution time does not deteriorate either.

In summary, for GAP problems of size (30x*n*) with *n* varying from 70 to 100, either execution time decreases using aspects or execution time is almost the same when compared to the version without aspects.

Table 10. Results for GAP problems with $m = 30$

n	Pr		Mean CPU time (s)				Difference API%			
		Without AOP	With AOP							
			Branching	Space generation	Choose subproblem	Cuts	Branching	Space generation	Choose subproblem	Cuts
70	1	24,879.20	24,692.8	24,964.94	24,803.68	25,043.72	0.75	-0.34	0.30	-0.66
70	2	45.46	45.50	43.74	43.47	43.54	-0.09	3.78	4.38	4.22
70	3	1,382.59	1,404.86	1,406.19	1,393.32	1,360.42	-1.61	-1.71	-0.78	1.60
70	4	2,380.38	2,337.97	2,353.87	2,337.55	2,358.38	1.78	1.11	1.80	0.92
80	1	145.38	155.64	146.27	145.39	145.52	-7.06	-0.61	-0.01	-0.10
80	2	30.02	29.74	28.99	28.79	28.81	0.95	3.43	4.10	4.03
80	3	10,033.54	10,556.30	10,356.56	10,272.24	10,406.58	-5.21	-3.22	-2.38	-3.72
80	4	465.22	430.78	427.87	425.14	432.96	7.40	8.03	8.62	6.93
90	1	202.27	200.91	204.82	203.46	203.29	0.67	-1.26	-0.59	-0.50
90	2	304.23	298.80	303.25	301.02	303.52	1.79	0.32	1.06	0.23
90	3	64.96	66.89	68.30	67.84	67.92	-2.97	-5.14	-4.43	-4.56
90	4	2,962.63	2,903.00	2,915.04	2,887.33	2,922.98	2.01	1.61	2.54	1.34
100	1	1,838.05	1,823.88	1,922.04	1,904.59	1,900.22	0.77	-4.57	-3.62	-3.38
100	2	187.01	180.66	183.13	181.75	181.87	3.40	2.07	2.81	2.75
100	3	413.66	415.27	405.13	402.03	401.93	-0.39	2.06	2.81	2.84
100	4	46,238.90	46,136.70	46,140.38	45,831.46	46,038.84	0.22	0.21	0.88	0.43

During experimentation, one of the questions asked was how much of the improvement in API% was caused by the transformation of the advices into inline functions by *AspectC++*. This was tested and the results showed that the difference in API% was not due to inline functions.

The original and aspect versions were compiled using the same compiler *Visual Studio .NET 2003 v7.0*. The profiler AQtime version 5.0—a 30-day trial version—was used to gain insight into the performance of COIN-OR with and without AOP. For each problem of size m, n a problem Pr was chosen from the one's shown in Tables, 8 and 9 with a higher API (%) value. Besides a tremendous increase in execution time (e.g., for a problem normally requiring an average of 25 min, the execution time with the profiler increases to 6 h), the profiler shows that on average the number of soft page faults without aspects doubles the number of soft page faults with AOP. The page faults occur when creating objects using the default constructor new and the operating system function *_heap_alloc_base* is invoked in *malloc* (*_nh_malloc_dbg*). The number of hard page faults is very low (2) in both versions.

Table 11. Execution time using a profiler for MKP problems with m = 20 and 25

m=20		Elapsed time			User time			User + Kernel time			Results in Table 8		
n	Pr	COIN-OR	Branching	API(%)	COIN-OR	Branching	API(%)	COIN-OR	Branching	API(%)	COIN-OR	Branching	API(%)
150	4	1.647,68	1.499,63	8,99	1.184,63	1.203,73	-1,61	1.248,90	1.274,17	-2,02	1.033,56	865,70	16,24
200	4	96,85	78,49	18,96	89,00	74,72	16,04	92,76	77,14	16,84	68,16	57,83	15,16
250	4	2.204,07	2.014,93	8,58	1.554,61	1.478,89	4,87	1.695,45	1.617,91	4,57	1.682,57	1.438,09	14,53
300	1	2.527,07	2.116,25	16,26	1.863,85	1.634,39	12,31	4.337,52	2.066,79	52,35	1.961,78	1.594,25	18,73
m=25		Elapsed time			User time			User + Kernel time			Results in Table 9		
n	Pr	COIN-OR	Branching	API(%)	COIN-OR	Branching	API(%)	COIN-OR	Branching	API(%)	COIN-OR	Branching	API(%)
150	4	298,23	308,53	-3,45	275,20	291,30	-5,85	288,44	301,91	-4,67	238,81	202,94	15,02
200	4	2.185,61	1.959,63	10,34	1.631,40	1.502,15	7,92	2.128,03	2.832,12	-33,09	1.530,76	1.258,30	17,80
250	3	4.571,48	4.066,42	11,05	3.433,58	3.037,89	11,52	3.468,27	3.145,51	9,31	3.773,57	3.083,01	18,30
300	2	11.169,48	9.436,20	15,52	7.880,67	6.946,14	11,86	7.771,94	7.189,27	7,50	9.105,36	7.420,92	18,50

This means that the compiler sometimes allocates the pages "better" when using AOP. Table 11 shows the Elapsed time, User time, and User plus Kernel time provided by the profiler. For most of the sizes and problems the API% changes significantly. In only one problem (m = 20; n = 200; Pr = 4) the value of API% is approximately the same as the one's shown in Tables 8 and 9.

The call graph for Pr = 4, m = 20, n = 200 in Fig. 7 shows the time consumed by *_heap_alloc_base* in the original version, and in Fig. 6, the partial call graph with aspect *Branching* is shown. The reduction in time for de *_heap_alloc_base* can be appreciated.

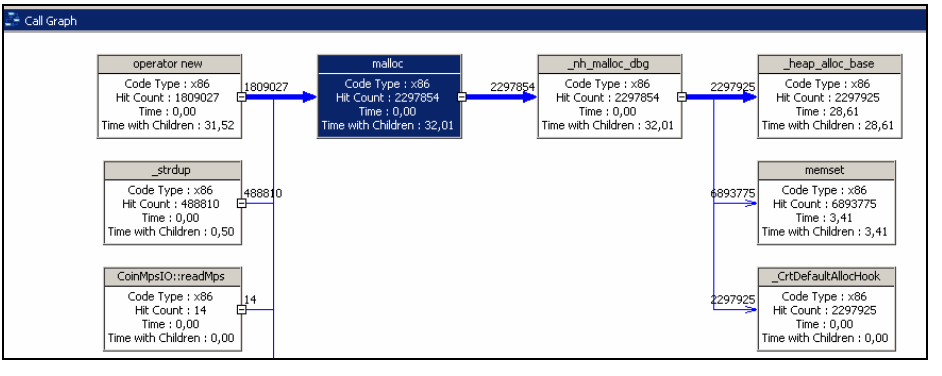


Fig. 6. Partial call graph using aspect *Branching*

The *_heap_alloc_base* function was identified when the method *createBranch* creates an instance of *SbbIntegerBranchingObject*, that is a *branching* object, *createBranch* is intercepted by aspect *Branching*.

In the aspect version, *_heap_alloc_base* consumes 38,29% (28.61 segs) of the total execution time (user time = 74.72 segs), the number of All Memory Page Faults counter calculated by the profiler is 599.025, # routine call is 787.146.261 and the number of page faults when executing *_heap_alloc_base* is 598.773. In the original version, *_heap_alloc_base* consumes 38,61% (34.36 segs) of the total execution time

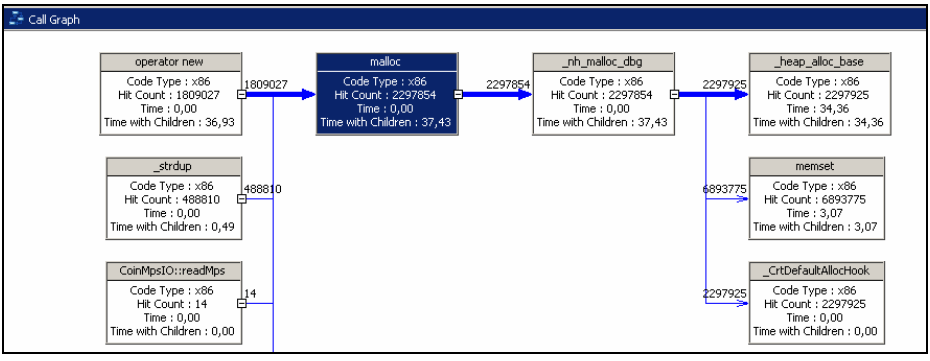


Fig. 7. Partial call graph for COIN-OR without aspects

(user time=89.00 segs), the number of All Memory Page Faults counter is 1.337.606, # routine call is 785.589.703 and the number of page faults when executing *_heap_alloc_base* is 1.337.362.

5 Conclusions and Future Work Direction

This paper proposes criteria to define concerns for the integer programming domain. We classified concerns according to three different criteria: functional requirements of algorithms, nonfunctional concerns, and concerns supporting a programmer's work. Furthermore, we have explored the use of AOP with two integer programming algorithms and compared their performance solving MKP and GAP problems using our refactored version, and to this end, we adapted the Monteiro and Fernandes' catalog defined for *AspectJ* modularization units to *C++*. Besides a cleaner implementation of B&B and B&C, the results indicate that execution time with AOP either decreases or stays more or less the same, specifically:

- A reduction of approximately 18% and 22% was obtained in the best case to solve the selected MKP problems with aspects *Branching* and *Cuts*. For the problems in the test, the execution time is always better with aspects.
- For GAP problems of the sizes used in the experimentation, the execution time does not deteriorate, in some cases it improves slightly.

These results show that using AOP in integer programming, at least in the algorithms B&B B&C, does not increase execution time.

Considering the results obtained with the profiler AQtime version 5.0, the compiler *Visual Studio .NET 2003 v7.0* sometimes allocates the pages "better" when using AOP thereby reducing the number of page faults by half. Even though our results might not be conclusive to explain why the AOP-refactored COIN-OR has performance benefits over the original code, they may be a motivation for a more in-depth exploration for other researchers.

Our aim was to provide a glimpse of the relevant results we have found in this ongoing research. In summary, three of the most significant lessons learned and reinforced in this investigation are:

1. The convenience of identifying categories of concerns prior to an implementation of refactoring with aspects.
2. The inadequacy or lack of existing tools for reverse engineering *C++* code and recognize behavior to refactor it with aspects. As there exists millions of lines of working *C++* code for solving real-life scientific problems these applications can be improved or at least customized using AOP; the lack of tools make this goal almost impossible to accomplish at this point.
3. The need to adapt and work with metric tools to find measures for specific situations.

Further investigation is needed and will eventually include new refactorings to be applied to other components in COIN-OR and an evaluation of structural characteristics of the refactored code. We hope that our experience will encourage the use of AOP in integer programming, a domain were the implementation techniques are usually conventional.

Acknowledgments. This work is supported by project No. PI-03-13-5198-2005, Consejo de Desarrollo Científico y Humanístico, Universidad Central de Venezuela.

References

1. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H.: Discussing Aspects of AOP. Communications of the ACM 44(10), 33–38 (2001)
2. Dijkstra, E.: On the Role of Scientific Thought. EWD 447. Springer, New York (1974); Selected Writings on Computing: A Personal Perspective, pp. 60–66. Springer, Heidelberg (1982)
3. Parnas, D.: On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM 15(12), 1053–1058 (1972)
4. Elrad, T., Filman, R., Bader, A.: Aspect-Oriented Programming. Communications of the ACM 44(10), 29–32 (2001)
5. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
6. Opdyke, W.: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. Ph.D. thesis. University of Illinois at Urbana-Champaign (1992)
7. Tourwé, T., Mens, T.: Identifying Refactoring Opportunities Using Logic Meta Programming. In: Proceedings 7th European Conference on Software Maintenance and Reengineering (CSMR), pp. 91–100. IEEE Computer Society, Los Alamitos (2003)
8. Fowler, M., Beck, K., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (2000)
9. Monteiro, M., Fernandes, J.: Towards a Catalog of Aspect-Oriented Refactoring. In: Proceedings Conference on Aspect-Oriented Software Development (AOSD), Chicago Illinois, USA, pp. 111–122. ACM Press, New York (2005)

10. Geoffrion, A., Marsten, R.: Integer Programming Algorithms: A Framework and State-of-the-Art Survey. Management Science 18(9), 465–491 (1972)
11. Salkin, H., Mathur, K., Robert, H.: Foundations of Integer Programming. North-Holland, Amsterdam (1989)
12. COmputational INfrastructure for Operations Research (COIN-OR) (2005), http://www.coin-or.org/index.html
13. https://projects.coin-or.org/Cbc/wiki/FAQ
14. Scientific Toolworks, Inc.: Understand for C++: User Guide and Reference Manual Version 1.4 (2005)
15. http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html
16. AspectC++ (2003), http://www.aspectc.org
17. Crema, A.: A contraction algorithm for the multiparametric integer linear programming problem. European Journal of Operational Research 101, 130–139 (1997)
18. Carnahan, J.: rng.cc Based on original C code by Steve Park & Dave Geyer (2003), http://www.cs.virginia.edu/~jcc5t/projects/oo_random/rng.cc
19. Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., Schröder-Preikschat, W.: A Quantitative Analysis of Aspect in the eCos Kernel. In: EuroSys Conference, Leuven, Belgium, April, pp. 191–204. ACM, New York (2006)

Appendix A

Table 12. AOP features in *AspectJ* and *AspectC++*

Concepts	*AspectC++*	*AspectJ*
aspect	Module implementing a crosscutting concern.	A crosscutting type encapsulating pointcuts, advices, and static crosscutting features.
advice	Is used either to specify code that should run when the *Joinpoints* specified by a *pointcut expression* are reached or to introduce a new method, attribute, or type to all *Joinpoints* specified by a *pointcut expression*.	Code that executes at each Joinpoint in a pointcut defined by before, after, and around declarations. While before advice is relatively unproblematic, there can be three interpretations of after advice: After the execution of a Joinpoint completes normally, after it throws an exception, or after it does either one. Methods, attributes, or types are added using *intertype* declarations.
Joinpoint	Denotes specific points in the base code where aspects can interfere.	A well-defined point in the execution of a program.
pointcut	Set of *Joinpoint* described by a *pointcut expression*.	A pointcut picks out Joinpoints and exposes some of the values in the execution context of those Joinpoints. There are several primitive pointcut designators and others can be named by the pointcut declaration.
introduction	If an advice is not recognized as being of a predefined kind (*before*, *after* and *around*), it is regarded as an introduction of a new method, attribute, or type to all Joinpoints in the pointcut.	Intertype declarations form *AspectJ*'s static crosscutting features, that is, code that may change the type structure of a program, by adding to or extending interfaces and classes with new fields, constructors, or methods.

Open Bisimulation for Aspects*

Radha Jagadeesan, Corin Pitcher, and James Riely

School of Computing, DePaul University, Chicago, IL 60604, USA
{rjagadeesan,cpitcher,jriely}@cs.depaul.edu

Abstract. We define bisimilarity for an aspect extension of the untyped lambda calculus and prove that it is sound and complete for contextual reasoning about programs. The language we study is very small, yet powerful enough to encode mutable references and a range of temporal pointcuts. We extend formal studies of Open Modules to this more general setting. Examples suggest that aspects are amenable to techniques developed for stateful higher-order programs. To our knowledge, this is the first study of coinductive reasoning principles for aspect programs.

1 Introduction

Aspects have emerged as a powerful tool in the design and development of systems [1, 2, 3, 4, 5, 6]. A standard example from the AspectJ tutorials suffices to introduce the basic vocabulary: Suppose class L realizes a useful library, and we want to obtain timing information about a method f of L. With aspects this can be done by writing *advice* specifying that, whenever f is called, the current time should be logged, f should be executed, and then the current time should again be logged. Aspects permit the profiling code to be localized in the advice, transferring the responsibility for coordinating the advice and base code to a compiler or runtime environment. In writing the logging advice, one must identify the pieces of code, using *pointcuts*, that need to be logged—in [7] this is called *quantification*. Furthermore, the developer of the library does not need explicit knowledge about advice that may be written in the future—this is called *obliviousness* in [7]. Aspect orientation is paradigm independent and has been realized in object-oriented [3, 8], imperative [9], and functional languages [10, 11].

Our focus in this paper is on the intersubstitutivity of programs written in an aspect-oriented extension of a functional language: when can one program fragment be substituted for another without altering the observable behavior of the program? A basic tool that has been used to address this question for other programming paradigms has been coinduction, in the form of bisimulation principles. While the origins of bisimulation trace back to concurrency theory (see [12, 13] for a comprehensive historical survey and detailed bibliography), bisimulation principles have proven to be quite useful to address program equality in several paradigms, e.g., higher-order languages (see [14, 15] for a detailed treatment with historical context), even in the presence of existential types [16] or state [17, 18], and object-oriented languages [19, 20].

* Based on an extended abstract published in AOSD 07, March 12–16, 2007, Vancouver, Canada, 1-59593-615-7/07/03.

A. Rashid and H. Ossher (Eds.): Transactions on AOSD V, LNCS 5490, pp. 72–132, 2009.

This paper brings aspect-based languages within the ambit of this technique. Our formal techniques and results suggest that aspects are no more intractable than stateful higher-order programs. In first-order languages with first-order references, when reasoning about programs, the environment has only two ways to interact with a program: either via global shared variables or by invoking the program (that can of course result in changes in encapsulated private state of the program). In higher-order languages with higher-order references, a program can also "leak" local state externally via higher-order mechanisms providing the environment a third way to interact with a program. Our results suggest that mechanisms that address this feature of higher-order languages with state may be adapted to an aspect framework with dynamic aspects.

Bisimulation. We study a core untyped lambda calculus, enhanced with aspects and named functions. Advice is first class in our calculus: it can be created and added dynamically while a program is running. The language can code mutable higher-order references and expressive pointcuts such as cflow and regular event patterns.

We describe a bisimulation principle based on a labeled transition system for aspect programs. We show that bisimulation is sound and complete for contextual equivalence.

We demonstrate the usability of the bisimulation principle via examples using the encoding of mutable variables—we show that several of the program equalities suggested by Meyer and Sieber [21] are validated by our bisimulation principle.

Application to Open Modules. Aspect–Aware Interfaces [22] enhance the usual signature information of modules with the pointcuts that are exported by the module and visible to the clients of the module. This enhancement of traditional signatures facilitates extra reasoning by providing bounds on the use of advice. An Open Module [23] delineates conditions about when it is permissible to replace the implementation of a module with another.

The formal treatment of Open Modules [23] only permits call pointcuts, whereas the implementation of Open Modules in AspectJ [24] also permits cflow pointcuts. Recent research has explored more expressive pointcut languages, such as those which match the entire computation history using regular patterns [25] or nested word languages [26, 27]. We use our bisimulation principle to bridge this expressiveness gap.

Our core calculus supports mechanisms to delimit the scope of the program where a function can be advised. We do this by providing named primitive pointcuts. Each function and advice declaration is associated with a primitive pointcut. Advice applies to a function only if its associated primitive pointcut is the same as that of the function. We use normal scoping mechanisms to control the knowledge of primitive pointcuts. The use of named primitive pointcuts as a separate construct permits the scope of the "advise" access to vary separately from the standard scope of direct access to the function reference.

This framework permits the use of our bisimulation principle to establish conditions under which implementations can be changed without affecting clients, even in the presence of dynamic aspects and an expressive collection of history-sensitive pointcuts.

Organization of this paper. After a discussion of related work, we present the core language in Sect. 3, including a definition of contextual equivalence and examples. In

Sect. 4, we describe the LTS and our notion of bisimulation; this section also contains examples that illustrate the use of the bisimulation. In Sect. 5, we state the foundational properties that hold. The bulk of the proofs are deferred to the appendices.

2 Related Work

Core calculi for aspect-based languages have been explored in a variety of settings. For example, [28, 29] are based on class-oriented calculi; in [30], a parametric description of a wide range of aspect languages is based on the object calculus [31]; and [32] integrates aspect and object-oriented languages. Our calculus builds on descriptions of aspects in higher-order functional languages [10, 11].

[33] provides a denotational semantics for a calculus with dynamic joinpoints, pointcut designators and advice. Our focus is on operational reasoning and proof rules. We refer the reader to [18] for a comparison of the operational and denotational approaches to stateful higher-order languages.

[34] provides the semantics of dynamic joinpoints by translating into a core functional language with simple matching features. Our approach complements this work by providing reasoning tools for a core functional language with aspects.

Formal static reasoning via type systems has been explored for functional [35] and object-oriented [36] aspect languages. Typing considerations are orthogonal to our primary focus, and we elide them to lighten the presentation.

Model-checking techniques have been explored to analyze the behavior of individual aspect programs [37, 38, 39]. Our paper is complementary to this research. Our study provides formal foundations for compositional proof principles that are of use in model-checking aspect programs. The utility of this approach is already suggested in [37].

There has also been research into facilitating reasoning by controlling obliviousness. For example, information flow methods have been used to create type systems that ensure that aspects do not affect the return value [40]—for some security applications, these superficially drastic sounding restrictions are appropriate. In this general spirit, albeit with less impact on obliviousness, the named primitive pointcuts of our calculus can be viewed as ways to control interference between aspects and between aspects and other code. Our primitive pointcuts are directly inspired by Open Modules [23] (see also [41]) and are a formal device to model some features of Aspect–Aware Interfaces [22]. There are two different views about where such names can originate: (a) as programming annotation, written by the programmer (a view arguably in tension with uninhibited obliviousness) or (b) a tool-derived annotation, derived from an analysis of the context of the program. In this paper, we do not take a viewpoint on this debate; instead, we focus on the support to reasoning that is afforded by such annotations.

Broadly speaking, bisimulation approaches to higher-order languages fall into the following main categories, depending on the kinds of tests that are permitted.

The first approach is usually termed applicative bisimulation. Some of the historical landmarks on this route are the initial definition of applicative bisimulation for lazy lambda calculus [42], the presentation using a labeled transition system [43], and a general method to show that applicative bisimulation is a congruence [44]. In this approach, two terms, say M_1 and M_2, that agree on convergence behavior are tested for bisimilarity

by providing them identical arguments and testing the resulting computation ($M_1\ N$ and $M_2\ N$) coinductively for bisimilarity. Extensions to account for imperative features were developed in [17].

In the second approach, the tests are enhanced. So, two lambda terms (say M_1 and M_2) are tested by providing them arguments that are derived from identical contexts (say $D[\cdot]$) with holes filled by bisimilar terms (say N_1 and N_2) and testing the resulting computation ($M_1\ D[N_1]$ and $M_2\ D[N_2]$) coinductively for bisimilarity. The complexity and number of tests is controlled by restricting attention to value contexts, i.e., $D[\cdot]$ such that $D[N_1]$ and $D[N_2]$ are values. [45] introduced this approach for untyped lambda-calculus with sealing/encryption, and [16] adopts it for polymorphic lambda-calculus with existential types. [18] develops this general framework for a higher-order language with imperative features. Class equivalences [46] and the object calculus [20] are also tackled by these methods. This general approach is now termed environmental bisimulation and its metatheory has been studied recently [47].

Our approach is inspired by open bisimulation [48], and ENF-bisimulation [49, 50]. In this approach, two lambda terms (say M_1 and M_2) are tested by providing them arguments that are symbolic names (say ϕ) and testing the resulting computation ($M_1\ \phi$ and $M_2\ \phi$) coinductively for bisimilarity. Any two terms with different symbols in the primary function position (say $\phi\ M$ and $\psi\ N$, where $\phi \neq \psi$ for arbitrary M and N) are considered different. Our approach to stateful programs is in particular closely related to the concurrently and independently developed treatment of sequential control and state [51] following this approach. Furthermore, the precise relationship of this style to game-theoretic semantics of programming languages [52, 53] has by now been formalized [54].

In comparison to applicative bisimulation, the more elementary congruence proofs of our approach suggest that our open-bisimulation based approach addresses stateful features more directly. In contrast to the environmental approaches to higher-order languages, our methods do not need to address the contextual closure of programs and equivalences of values in this closure. However, the price paid by our approach is the explicit maintenance of extra contexts and transitions for book-keeping mechanisms. We develop congruence results and bisimulation-upto results to lighten this burden. In Sect. 4, we present a detailed comparison of our definitions with the two approaches.

In summary, the examples in this paper suggest that our treatment is good enough to capture and formalize intuitions crystallized by observation of the source code. However, we do not have any results that support the (semi-)automatic derivation of witnessing relations. That investigation remains open to future study.

3 Language

Our calculus builds on descriptions of aspects in higher-order functional languages [10, 11]. Advice may be loaded dynamically; several recent aspect language implementations support such dynamic aspects, e.g., [55]. Primitive pointcuts are named and scoped: a programmer may limit the scope over which a function is advisable by controlling the scope of the associated primitive pointcut. In this respect, our language has some of the expressiveness of the module language of [23] in a simpler setting.

Each function declaration is associated with a primitive pointcut and advice applies to a function only if its associated primitive pointcut is that of the function. One may view possession of the name of a function as a form of *read access* and possession of the primitive pointcut of a function as a form of *write access*. We formalize this intuition when encoding references in Example 7.

The language is an untyped lambda calculus extended with function declarations in the style of ML and with advice over declared functions. The difference between abstractions and declared functions can be detected contextually. For example, consider $(\lambda\_.0)$ and $(\textbf{fun}\ \mathsf{f@p} = \lambda\_.0\,;\mathsf{f})$, which declares f at primitive pointcut p and returns f. The first expression results immediately in an abstraction. The second results in the name f, which is only resolved to an abstraction when applied. The difference is observable when the primitive pointcut p is used to declare advice, as, for example, in the context $(\textbf{adv}\ \mathsf{p} = \lambda\_.1\,;[\text{–}]\ ())$; here [–] is the "hole" to be filled by a term. The context declares advice at p then applies the hole to the unit value; evaluation results in 0 when the hole is filled with $(\lambda\_.0)$, but 1 when filled with $(\textbf{fun}\ \mathsf{f@p} = \lambda\_.0\,;\mathsf{f})$. A function declared at a bound primitive pointcut is unadvisable outside the scope of the binder; thus, $(\lambda\_.0)$ and $(\textbf{pcd}\ \mathsf{p}\,;\textbf{fun}\ \mathsf{f@p} = \lambda\_.0\,;\mathsf{f})$ are contextually indistinguishable.

In the rest of this section, we formalize the syntax (Sect. 3.1) and dynamics (Sects. 3.2 and 3.3) of this core calculus. Section 3.4 defines contextual equivalence. Section 3.5 provides simple examples to illustrate the definitions. Section 3.6 discusses Open Modules and temporal pointcuts.

3.1 Syntax

We divide names into two countably infinite and mutually disjoint sets: variables and primitive pointcuts. In this study, primitive pointcuts are second-class entities; we discuss the motivation for this decision in Example 11.

SYNTAX

$f,g,h,x,y,z,\phi,\psi,\theta$	Variable Names
p,q,r	Primitive Pointcut Descriptors
$A,B ::=$	Declarations
$\quad \mathsf{pcd}\ p$	$\quad$ Primitive Pointcut Descriptor ($dn = \{p\}$)
$\quad \mathsf{fun}\ f\mathsf{@}p = U$	$\quad$ Function ($dn = \{f\}$)
$\quad \mathsf{adv}\ p = \lambda z.U$	$\quad$ Advice ($dn = \{\}$, z bound in U)
$U,V,W ::=$	Values
$\quad x$	$\quad$ Variable
$\quad \lambda x.M$	$\quad$ Abstraction (x bound in M)
$M,N,L ::=$	Terms
$\quad U$	$\quad$ Value
$\quad A\,;M$	$\quad$ Declaration ($dn(A)$ bound in M)
$\quad \mathsf{let}\ x = M\,;N$	$\quad$ Sequence (x bound in N)
$\quad U\ V$	$\quad$ Application

The name declared by a declaration is given by the function dn, defined in the syntax table above. We assume the usual notion of free names, recovered by the function fn. While recursive uses of f are not allowed in $\mathsf{fun}\ f\mathtt{@}p = U$, this is not a limitation; see Example 6. We identify terms up to renaming of bound names and write $M[x := U]$ for the capture-avoiding substitution of U for x in M, and similarly for $M[p := q]$. Thus, $\mathsf{pcd}\ p\,;M$ is identical to $\mathsf{pcd}\ q\,;M[p := q]$ for any $q \notin fn(M)$.

We use the following discipline for variable names, when feasible. (The distinctions, while useful in many cases, are blurred when discussing congruence.)

- z is used for *proceed variables* bound in the body of advice;
- x-y are used for variables bound in abstractions and let-expressions, other than as a proceed variable;
- f-h are used for variables bound by function declarations; and
- ϕ-θ are used for free function variables.

Variables x-y are resolved, in the standard way, during evaluation (Sect. 3.3). Variables z and f-h are resolved during function lookup (Sect. 3.2). The variables ϕ-θ are unresolvable; these are used in the LTS semantics (Sect. 4).

In examples, we use the unit value $()$, booleans (tru and fls), integers and pairs of values. These represent the standard Church encodings [56] (where $()$ is any value); thus 0, tru and fst are encoded as the combinator $(\lambda x.\lambda y.x)$, fls and snd are $(\lambda x.\lambda y.y)$, and 1 is $(\lambda x.\lambda y.y\ x)$. The Church-encoded constants may not satisfy all the equations one would like; we use them only for parsimony. Primitive constants may be added to the definition of the labeled transition system in Sect. 4 in the standard way [57]. We also use other well-known combinators, such as the divergent term $\Omega \triangleq (\lambda x.x\,x)\ (\lambda x.x\,x)$.

We use syntax sugar for application in the style of Moggi [58]; for example, $(M\,N) \triangleq (\mathsf{let}\ x = M\,;\ \mathsf{let}\ y = N\,;\ x\,y)$, where $x \notin fn(N)$. We adopt the same convention for operators on booleans, naturals and pairs. We write $\_$ for a bound variable that does not occur free in its scope; we abbreviate $(\mathsf{let}\ \_ = M\,;N)$ as $(M\,;N)$ and $(\lambda\,\_.M)$ as $(\lambda.M)$.

In examples, we sometimes write $(\mathsf{fun}\ f = U)$ as shorthand for $(\mathsf{pcd}\ p\,;\mathsf{fun}\ f\mathtt{@}p = U)$, when p is not of interest. We also occasionally write declarations as terms, with the meaning that A, as a term, abbreviates $(A\,;())$.

3.2 Lookup

In this subsection, we describe function lookup, which determines the body of an advised function from a declaration sequence (notation $\vec{A}(f) = U$). We write $\vec{A}$ for *declaration sequences*, with "$\cdot$" representing the empty sequence and "$;$" the element separator. An *evaluation configuration* is a pair of a declaration sequence and a term, written $\vec{A}/M$. To motivate the formal definition of lookup, we first present a few examples.

Example 1. Let $\vec{\mathsf{A}}$ be defined as follows.

$$\begin{array}{lll} \vec{\mathsf{A}} = \mathbf{adv}\ \mathsf{p} = \lambda z.\mathsf{V}\,; & & \mathsf{V} = \lambda y.(z\,y) + 1 \\ \quad \mathbf{fun}\ \mathsf{f@p} = \mathsf{W}\,; & \text{where} & \mathsf{W} = \lambda.5 \\ \quad \mathbf{adv}\ \mathsf{p} = \lambda z.\mathsf{U} & & \mathsf{U} = \lambda x.(z\,x) * 3. \end{array}$$

When one looks up f in the context of $\vec{\mathsf{A}}$, the result is

$$\vec{\mathsf{A}}(\mathsf{f}) = \mathsf{U}\big[\mathsf{z} := \mathsf{V}[\mathsf{z} := \mathsf{W}]\big] = \lambda \mathsf{x}.((\lambda \mathsf{y}.((\lambda .5)\,\mathsf{y}) + 1)\,\mathsf{x}) * 3.$$

The top-level term is U: the last (or most recently) declared advice which affects f (via the primitive pointcut p). The proceed variable z of U is bound to the rest of the advice which effects f, in this case V. Substitutions layer in this way to the last piece of advice, which proceeds to the function body, in this case W.

Evaluation of $\mathsf{f}\,()$ proceeds as follows.

$$\begin{aligned} \cdot/\vec{\mathsf{A}}\,;\mathsf{f}\,() &\twoheadrightarrow \vec{\mathsf{A}}/((\lambda \mathsf{y}.((\lambda .5)\,\mathsf{y}) + 1)\,())*3 \\ &\twoheadrightarrow \vec{\mathsf{A}}/(((\lambda .5)\,()) + 1)*3 \\ &\twoheadrightarrow \vec{\mathsf{A}}/18. \end{aligned}$$

Lookup is a partial function on names. For example, using the declarations above, $\vec{\mathsf{A}}(\mathsf{g})$ is undefined, and thus the evaluation configuration $\vec{\mathsf{A}}/\mathsf{g}()$ is stuck. □

Example 2. Note that advice may ignore the definition of the underlying function or of other advice—both referenced via z. As an example, consider

$$\begin{array}{lll} \vec{\mathsf{B}} = \textbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{V}\,; & & \mathsf{V} = \lambda .7 \\ \quad \textbf{fun}\ \mathsf{f@p} = \mathsf{W}\,; & \text{where} & \mathsf{W} = \lambda .5 \\ \quad \textbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U} & & \mathsf{U} = \lambda \mathsf{x}.(\mathsf{z}\,\mathsf{x}) * 3. \end{array}$$

In this case

$$\vec{\mathsf{B}}(\mathsf{f}) = \mathsf{U}\big[\mathsf{z} := \mathsf{V}[\mathsf{z} := \mathsf{W}]\big] = \lambda \mathsf{x}.((\lambda .7)\,\mathsf{x}) * 3$$

and evaluation of $\mathsf{f}\,()$ proceeds as follows.

$$\cdot/\vec{\mathsf{B}}\,;\mathsf{f}\,() \twoheadrightarrow \vec{\mathsf{B}}/((\lambda .7)\,())*3 \twoheadrightarrow \vec{\mathsf{B}}/21$$ □

Lookup is defined using two auxiliary functions: *body* and *advise*. Whereas we identify terms up to renaming of bound names, the same does not hold for names declared in a declaration sequence. (This treatment is motivated by the definition of *body*, by which a primitive pointcut may escape its scope.) Instead, we require that declaration sequences be *well formed*, i.e., that each name be declared at most once. For example, the declaration sequence $(\textbf{fun}\ \mathsf{f@p} = \mathsf{U}\,;\textbf{fun}\ \mathsf{f@q} = \mathsf{V})$ is not well formed.

Definition 3 *(Wellformedness).* A declaration sequence "$\vec{A}\,;B$" is *well formed* if $dn(B)$ does not occur in $\vec{A}$ and $\vec{A}$ is well formed. The empty sequence is also well formed.

An evaluation configuration $\vec{A}/M$ is *well formed* if $\vec{A}$ is well formed. □

Note that in a well-formed evaluation configuration $\vec{A}/M$, there may be names that occur free in M that are not declared in $\vec{A}$ (cf. Example 1).

The partial function $body(f, \vec{A})$ is defined whenever f is declared in $\vec{A}$; when defined, *body* returns both the value of the function and the primitive pointcut at which f is declared in $\vec{A}$.

$$\begin{aligned} body(f, \cdot) &\triangleq \text{undefined} \\ body(f, \mathsf{pcd} \cdots ; \vec{A}) &\triangleq body(f, \vec{A}) \end{aligned}$$

$$\begin{aligned} body(f, \mathsf{fun}\ f@p = U\,;\vec{A}) &\triangleq \langle p, U\rangle \\ body(f, \mathsf{fun}\ g@p = U\,;\vec{A}) &\triangleq body(f,\vec{A}),\ \text{where}\ f \neq g \\ body(f, \mathsf{adv}\ \cdots\,;\vec{A}) &\triangleq body(f,\vec{A}) \end{aligned}$$

The total function $advise(p, U, \vec{A})$ returns a value that applies to U the advice declared in $\vec{A}$ for p.

$$\begin{aligned} advise(p, U, \cdot) &\triangleq U \\ advise(p, U, \mathsf{pcd}\ q\,;\vec{A}) &\triangleq advise(p, U, \vec{A}),\ \text{where}\ p \neq q \\ advise(p, U, \mathsf{fun}\ \cdots\,;\vec{A}) &\triangleq advise(p, U, \vec{A}) \\ advise(p, U, \mathsf{adv}\ p = \lambda z.V\,;\vec{A}) &\triangleq advise(p, V[z := U], \vec{A}) \\ advise(p, U, \mathsf{adv}\ q = \lambda z.V\,;\vec{A}) &\triangleq advise(p, U, \vec{A}),\ \text{where}\ p \neq q \end{aligned}$$

Finally, the partial function $\vec{A}(f)$ is defined as follows.

$$\vec{A}(f) \triangleq \begin{cases} advise(p, V, \vec{A}) & \text{if } body(f,\vec{A}) = \langle p, V\rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

3.3 Dynamics

Following [59], reduction is defined using *evaluation contexts*, defined as follows.

$$\mathscr{E}, \mathscr{F}, \mathscr{G} \ ::=\ [-] \ \mid\ \mathsf{let}\ x = \mathscr{E}\,;N$$

As usual, $[-]$ is the "hole" to be filled by a term. Reduction is defined inductively as a binary relation between well-formed configurations, using four axiom schemas.

REDUCTION $(\vec{A}/M \longrightarrow \vec{A}'/M')$

$\vec{A}/\mathscr{E}[B\,;M]$	$\longrightarrow \vec{A}\,;B/\mathscr{E}[M]$	if $dn(B) \notin dn(\vec{A}) \cup fn(\mathscr{E})$
$\vec{A}/\mathscr{E}[\mathsf{let}\ x = U\,;N]$	$\longrightarrow \vec{A}/\mathscr{E}\left[N[x := U]\right]$	
$\vec{A}/\mathscr{E}[f\ V]$	$\longrightarrow \vec{A}/\mathscr{E}\left[U\ V\right]$	if $\vec{A}(f) = U$
$\vec{A}/\mathscr{E}[(\lambda x.M)\ V]$	$\longrightarrow \vec{A}/\mathscr{E}\left[M[x := V]\right]$	

The first axiom is structural, regulating the scope of declarations. Recall that we allow renaming of bound variables in terms, but not declaration sequences. Since the set of names is infinite, evaluation configurations of the form $\vec{A}/\mathscr{E}[B\,;M]$ may always reduce, fixing a "fresh" name for $dn(B)$.

The axiom for sequencing is standard, reducing $\mathsf{let}\ x = M\,;N$ only when M is a value. The use of contexts makes structural rules unnecessary. For example, if $\vec{A}/\mathscr{E}[f\ V] \longrightarrow \vec{A}/U\ V$, then $\vec{A}/\mathsf{let}\ x = \mathscr{E}[f\ V]\,;N \longrightarrow \vec{A}/\mathsf{let}\ x = \mathscr{E}[U\ V]\,;N$ by choosing the context $\mathscr{E}' = \mathsf{let}\ x = \mathscr{E}\,;N$.

There are three possibilities for an application $\vec{A}/\mathscr{E}[U\ V]$: (1) If U is a function name f and $\vec{A}(f)$ is defined, then evaluation proceeds to $\vec{A}/\mathscr{E}[\vec{A}(f)\ V]$; (2) If U is an abstraction then evaluation proceeds call-by-value using U; this is the standard beta-reduction axiom; and (3) Otherwise evaluation is stuck.

Write $\longrightarrow\!\!\!\!\!\rightarrow$ for the reflexive transitive closure of $\longrightarrow$.

Example 4 Consider the following evaluation configuration.

$$\cdot\,/\mathbf{fun}\ \mathsf{id@p} = \lambda \mathsf{x}.\mathsf{x}\,;\ \mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\lambda \mathsf{y}.\mathsf{z}\ \mathsf{z}\ \mathsf{y}\,;\ (\lambda \mathsf{f}.\mathsf{f}\ 5)\ \mathsf{id}.$$

Using the axiom for declarations twice this reduces to

$$\twoheadrightarrow \mathbf{fun}\ \mathsf{id@p} = \lambda \mathsf{x}.\mathsf{x}\,;\ \mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\lambda \mathsf{y}.\mathsf{z}\ \mathsf{z}\ \mathsf{y}/(\lambda \mathsf{f}.\mathsf{f}\ 5)\ \mathsf{id}$$

which the axiom for application further reduces to

$$\longrightarrow \mathbf{fun}\ \mathsf{id@p} = \lambda \mathsf{x}.\mathsf{x}\,;\ \mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\lambda \mathsf{y}.\mathsf{z}\ \mathsf{z}\ \mathsf{y}/\mathsf{id}\ 5.$$

Note that id is treated as a pure name when passed as an argument; it is only resolved at the point of application, where the axioms for lookup and beta-reduction yield

$$\begin{aligned}
&\mathbf{fun}\ \mathsf{id@p} = \lambda \mathsf{x}.\mathsf{x}\,;\ \mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\lambda \mathsf{y}.\mathsf{z}\ \mathsf{z}\ \mathsf{y}/(\lambda \mathsf{y}.(\lambda \mathsf{x}.\mathsf{x})\ (\lambda \mathsf{x}.\mathsf{x})\ \mathsf{y})\ 5\\
\longrightarrow\ &\mathbf{fun}\ \mathsf{id@p} = \lambda \mathsf{x}.\mathsf{x}\,;\ \mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\lambda \mathsf{y}.\mathsf{z}\ \mathsf{z}\ \mathsf{y}/(\lambda \mathsf{x}.\mathsf{x})\ (\lambda \mathsf{x}.\mathsf{x})\ 5\\
\longrightarrow\ &\mathbf{fun}\ \mathsf{id@p} = \lambda \mathsf{x}.\mathsf{x}\,;\ \mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\lambda \mathsf{y}.\mathsf{z}\ \mathsf{z}\ \mathsf{y}/(\lambda \mathsf{x}.\mathsf{x})\ 5\\
\longrightarrow\ &\mathbf{fun}\ \mathsf{id@p} = \lambda \mathsf{x}.\mathsf{x}\,;\ \mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\lambda \mathsf{y}.\mathsf{z}\ \mathsf{z}\ \mathsf{y}/5.
\end{aligned}$$

□

3.4 Contextual Equivalence

Contextual equivalence is defined with respect to a primitive notion of observation; two terms are related if they yield the same observations in all contexts. Following [60, 61], we assume a distinguished function name signal and take a call to this function to be a primitive observation.

Definition 5. A *(general) context* is any term with a single hole:

$$\mathscr{C} ::= [\text{–}] \mid A\,;\mathscr{C} \mid \mathsf{let}\ x = \mathscr{C}\,;N \mid \mathsf{let}\ x = M\,;\mathscr{C} \mid \mathscr{C}\ N \mid M\ \mathscr{C}.$$

Write $M\!\downarrow$ if $M \twoheadrightarrow \mathscr{E}[\mathsf{signal}\ U]$ for some evaluation context $\mathscr{E}$ and value U. For terms M and N in which signal does not occur, define $M \leqq N$ if for every context $\mathscr{C}$,

$$\mathscr{C}[M]\!\downarrow \text{ implies } \mathscr{C}[N]\!\downarrow.$$

Two terms M and N are *contextually equivalent* ($M \equiv N$) if $M \leqq N$ and $N \leqq M$. □

For lambda calculi, the primitive observation for contextual equivalence is usually taken to be termination [62]. Because our language includes side effects, however, a finer observation is necessary for completeness (Appendix E). For example, there is no context which, on the basis of termination alone, can distinguish $((\mathbf{adv}\ \mathsf{p} = \lambda.\lambda.1)\,;\mathsf{f}\,()\,;\Omega)$ from $((\mathbf{adv}\ \mathsf{p} = \lambda.\lambda.2)\,;\mathsf{f}\,()\,;\Omega)$, since both terms are divergent. (Ω is defined toward the end of Sect. 3.1.) Using signal, these can be distinguished by the context

$$(\mathbf{fun}\ \mathsf{g@p} = \lambda.0)\,;(\mathbf{fun}\ \mathsf{f} = \lambda.\mathbf{if}\ \mathsf{g}\,() == 1\ \mathbf{then}\ \mathsf{signal}\,()\ \mathbf{else}\ \Omega)\,;[\text{–}].$$

3.5 Simple Examples

Example 6 *(Recursive use of function names).* Note that in a function definition $(\mathsf{fun}\ f@p = U)$, the function name f is not bound in U. Recursive uses of f in U may be accommodated by writing the definition as

$$(\mathsf{fun}\ f@p = V)\,;\,(\mathsf{adv}\ p = \lambda\,.\,U),$$

where V is "dummy" value; the advice on f does not proceed and therefore V is ignored.

For example, one definition of Ω is "poisonpill ()", where poisonpill is the divergent function "**fun** poisonpill = λ . poisonpill ()". This may be written in our language as

$$(\mathbf{fun}\ \mathsf{poisonpill@p} = \lambda\,.\,()\,)\,;\,(\mathbf{adv}\ \mathsf{p} = \lambda\,.\,\lambda\,.\,\mathsf{poisonpill}\ ()\,),$$

where we have chosen "$\lambda\,.\,()$" as the dummy value. □

Example 7 *(References).* We show how to code ML-style references as syntax sugar in the language of terms. The example demonstrates the unsurprising fact that dynamically loaded advice is a form of mutability.

We model references as a pair of functions, where the first is used for reading and the second for writing; the first is locally advisable, whereas the second is not. If p and f do not occur free in U, then define the following.

$$\begin{aligned}\mathsf{ref}\ U &\triangleq \mathsf{pcd}\ p\,;\,(\mathsf{fun}\ f@p = \lambda\,.\,U)\,;\,(f, \lambda x.\mathsf{adv}\ p = \lambda\,.\,\lambda\,.\,x)\\ !U &\triangleq (\mathsf{fst}\ U)\,()\\ U := V &\triangleq (\mathsf{snd}\ U)\,V\,;\,()\end{aligned}$$

We can code the imperative factorial as

$$\begin{aligned}&\mathbf{fun}\ \mathsf{fac} = (\lambda \mathsf{x}.(\mathbf{let}\ \mathsf{y} = \mathsf{ref}\ 1)\,;\,(\mathbf{fun}\ \mathsf{loop} = \mathsf{U})\,;\,\mathsf{loop}\ \mathsf{x}),\ \text{ where}\\ &\mathsf{U} = \lambda \mathsf{x}.\mathbf{if}\ (\mathsf{x} \leq 1)\ \mathbf{then}\ (!\mathsf{y})\ \mathbf{else}\ (\mathsf{y} := !\mathsf{y} * \mathsf{x}\,;\,\mathsf{loop}\ (\mathsf{x} - 1)).\end{aligned}$$

Eliding the definitions of fac, loop, and p, fac 2 evaluates as

$$\begin{aligned}\cdot/\mathsf{fac}\ 2 &\twoheadrightarrow \mathbf{fun}\ \mathsf{f@p} = \lambda\,.\,1/\mathsf{loop}\ 2\\ &\twoheadrightarrow \mathbf{fun}\ \mathsf{f@p} = \lambda\,.\,1\,;\,\mathbf{adv}\ \mathsf{p} = \lambda\,.\,2/\mathsf{loop}\ 1\\ &\twoheadrightarrow \mathbf{fun}\ \mathsf{f@p} = \lambda\,.\,1\,;\,\mathbf{adv}\ \mathsf{p} = \lambda\,.\,2/\mathsf{f}\ ()\\ &\twoheadrightarrow \mathbf{fun}\ \mathsf{f@p} = \lambda\,.\,1\,;\,\mathbf{adv}\ \mathsf{p} = \lambda\,.\,2/2.\end{aligned}$$

The result is 2, as expected. □

Example 8 *(Contexts may need to test a value more than once).* It is important that a context may store a value and test it more than once. For example, the terms $(\lambda\,.\,0)$, which always returns 0, and

$$\mathbf{let}\ \mathsf{b} = \mathsf{ref}\ \mathsf{tru}\,;\,(\lambda\,.\mathbf{if}\ !\mathsf{b}\ \mathbf{then}\ (\mathsf{b} := \mathsf{fls}\,;\,0)\ \mathbf{else}\ 1),$$

which returns 0 exactly once, can be distinguished by the context

$$\mathbf{let}\ \mathsf{x} = [-]\,;\,\mathsf{x}\,()\,;\,\mathbf{if}\ \mathsf{x}\,() = 0\ \mathbf{then}\ \mathsf{signal}\,()\ \mathbf{else}\ \Omega.$$

□

Example 9 *(Contexts can observe advice order).* To show some of the subtleties of contextual reasoning, here is an example where a context inserts itself in the middle of an advice list.

$$\mathscr{E} = \textsf{let}\ \mathsf{x} = [\_]\,;\ \textsf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{V}\,;\ \mathsf{x}\,()$$

Consider

$$\mathscr{E}[\textsf{fun}\ \mathsf{f@p} = \lambda \mathsf{x}.0\,;\ \textsf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U}_1\,;\ (\lambda\,.(\textsf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U}_2\,;\ \mathsf{f}\ 0))]$$

which evaluates to

$$\cdots\,;\ \mathsf{U}_2\big[\mathsf{z} := \mathsf{V}[\mathsf{z} := \mathsf{U}_1[\mathsf{z} := \lambda \mathsf{x}.0]]\big].$$

Here the context has inserted the advice V between two bits of user advice U_2 and U_1. Using $\mathsf{V} = (\lambda \mathsf{x}.\textsf{if}\ \mathsf{x} = 1\ \textsf{then}\ \mathsf{signal}()\ \textsf{else}\ \Omega)$, the context can distinguish the following pairs of advice, which cannot be distinguished without advising p.

$$\mathsf{U}_1 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+0) \qquad \mathsf{U}'_1 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+1)$$
$$\mathsf{U}_2 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+1) \qquad \mathsf{U}'_2 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+0)$$

When composed, both pairs of advice add 3 to the function's argument. The advice V observes the intermediate result of the computation. □

Example 10 *(Indistinguishability of functions).* Functions with the same body declared at the same primitive pointcut are indistinguishable. The following terms are contextually equivalent for any M.

$$\textsf{fun}\ \mathsf{f@p} = \lambda \mathsf{x}.M\,;\ \textsf{fun}\ \mathsf{g@p} = \lambda \mathsf{x}.M\,;\ (\mathsf{f}, \mathsf{g})$$
$$\textsf{fun}\ \mathsf{h@p} = \lambda \mathsf{x}.M\,;\ (\mathsf{h}, \mathsf{h})$$

One can prove the equivalence using the method presented in the Sect. 4. □

3.6 Open Modules and Temporal Pointcuts

In this subsection, we consider encodings of Open Modules as proposed by Aldrich [23]. Open Modules extend ML-style modules to support two methods for controlling aspects:

- a distinction between internal and external function calls—only external calls are advisable from outside the module; and
- explicit pointcut declaration in module interfaces—only declared pointcuts may be used externally.

The first feature is handled in the operational semantics of [23] by renaming the function and creating a fresh declaration of the original name to invoke it. This kind of renaming can be achieved in compilation; here, we write programs directly in the form such a compiler would produce.

The second feature is more subtle, and we address it in two ways.

- We provide distinct binders for functions and primitive pointcuts; these may be viewed, respectively, as read and write capabilities, which may be handled independently. We treat primitive pointcuts as second class, since they are intended to delimit the static scope of mutability.
- We allow dynamically loaded advice. In addition to encoding state (Example 7), dynamically loaded advice allows us to create expressive "pointcuts" and to communicate them selectively as abstractions (Examples 13 and 14).

Example 11 *(Open Modules).* To get a sense of our approach, consider a concrete example: a math module with one advisable function fac. Internal and external calls to fac are distinguished so that only external calls may be advised.

```
module type MATH = sig
  val fac : int → int
  pointcut pfac : int → int
end;;
module Math : MATH = struct
  let rec fac = fun n → if n<1 then 1 else n*fac(n-1)
  pointcut pfac = call(fac)
end;;
open Math;;
let main = fun _→ fac 5;;
```

We view the module as providing two functions: the first is fac itself and the second is the pointcut pfac. A call to pfac will place advice on external calls to fac. In a module system, the calls to pfac occur in the compiler rather than at runtime, but this phase distinction is an implementation convenience rather than a necessity.

The example can be coded in our language as follows. (Recall from Section 3.1 that (**fun** f = U) is syntactic sugar for (**pcd** p ; **fun** f@p = U), where p is a fresh primitive pointcut descriptor.)

```
fun Math = λ.
  fun fac′ = λn.if n<1 then 1 else n*fac′(n-1);
  pcd pfac′;
  fun fac@pfac′ = fac′;
  (fac, λy.adv pfac′ = λz.λx.y z x);
let (fac, pfac) = Math ();
fun main = λ.fac 5
```

The functions fac and pfac, recovered from Math, correspond exactly to the functions provided by the module above. For example, to count the number of calls to fac using a reference c, one might proceed as follows.

```
let c = ref 0;
pfac (λz.λx.c := !c+1 ; z x)
```

□

Remark 12 *(Modularity results).* In the above example, whereas fac is publicly advisable, fac′ is private to Math. To see that internal calls to fac′ are unadvisable, note that one could exchange the body of fac′ given here with that from Example 7 and the result would be contextually equivalent to the original (compare with Sect. 5.2 of [23]). In fact, the following general result holds. Let

$$\mathscr{C} = \textbf{pcd}\ \mathsf{p}\,;\ \textbf{fun}\ \mathsf{f@p} = [\text{–}]\,;\ \mathsf{f}$$
$$\mathscr{D} = \textbf{fun}\ \mathsf{g@q} = [\text{–}]\,;\ \mathsf{M}$$

for any M. Then,

$$\mathscr{C}[\mathsf{U}] \equiv \mathscr{C}[\mathsf{V}] \text{ implies } \mathscr{D}\big[\mathscr{C}[\mathsf{U}]\big] \equiv \mathscr{D}\big[\mathscr{C}[\mathsf{V}]\big].$$

This follows immediately from the fact that $\equiv$ is a congruence (Sect. 5). This general result allows any function to be defined in such a way that external calls are advisable, while internal ones are not. The remarkable power of contextual reasoning guarantees that the internal body can be substituted with any locally equivalent body without effecting the overall observable behavior. □

The previous encoding can be extended to richer pointcut languages, while still maintaining the modularity results.

Example 13 *(cflow).* The AspectJ pointcut call(f) && cflow(g) detects calls to f in the context of a call to g. Such a pointcut is exported from the following module.

```
fun FcflowG = λ.
  pcd pf; fun f@pf = ··· ;
  pcd pg; fun g@pg = ··· ;
  let b = ref fls; // call to g active
  adv pg = λz.λx.let b′ = !b; b := tru; let y = z x; b := b′; y;
  (f, g, λy.adv pf = λz.λx.if !b then y z x else z x) ;
let (f, g, pf_cflow_g) = FcflowG () ;
```

The local boolean reference b is used to record whether a call to g is active. Whenever g is called, the advice at pg sets b to tru, proceeds to the body of g, and then resets b. Whenever f is called, the advice at pf first checks b before proceeding to the body of f.

A user may advise "f in the context of g", by calling the pf_cflow_g with advice $\lambda z.\lambda x.\cdots$. However, no other pointcuts are exposed. This generalizes the technique of Aldrich, and indeed, the congruence results (c.f. Remark 12) apply equally to such terms. □

Nested word languages [26, 27] are a subset of context-free languages with good closure properties that capture sensitivity to both the call stack (as in cflow) and other history (as in regular patterns [25]). Pointcuts based on nested word languages arise naturally in examples in security (access control) and document processing (XML transducers). Since the operational semantics of nested word languages pushes exactly one stack symbol upon reading a call symbol and pops exactly one stack symbol upon reading a return symbol, such pointcut languages are addressable by implementation methods developed for cflow and regular patterns. The next example illustrates the ingredients of a translation from temporal pointcuts specified via nested word languages.

Example 14 *(History-sensitive access control).* Abadi and Fournet [63] argue for history-sensitive access control mechanisms more expressive than the stack inspection mechanisms found in Java and C#. For example, consider a policy stating that advice on a sensitive function rm (e.g., for file deletion) should be executed only if an (untrusted) function un has never been invoked in the past, *and* no call to f is still active. This policy for an access control failure is specified as a nested word language over symbols drawn from calls to, and returns from, un, rm, and f. Using EBNF syntax, let balanced and opencalls be defined as follows.

$$\begin{aligned} \mathsf{balanced} &::= \big((\mathit{call}(.)\ \mathsf{balanced}\ \mathit{ret}(.))\big)* \\ \mathsf{opencalls} &::= \big(\mathsf{balanced} \mid \mathit{call}(.)\big)* \end{aligned}$$

The property of interest can then be written as

$$\big(\underbrace{(.*\ \mathit{call}(\mathsf{un})\ .*)}_{\mathsf{un\ called}} \mid \underbrace{(\mathsf{opencalls}\ \mathit{call}(\mathsf{f})\ \mathsf{opencalls})}_{\mathit{call}(\mathsf{f})\ \mathsf{active}}\big)\ \mathit{call}(\mathsf{rm}).$$

Following Example 13, we can export a pointcut matching the negation of this property of the call history.

```
fun Hsac = λ.
  pcd pun; fun un@pun = ··· ;
  pcd pf ; fun f@pf = ··· ;
  pcd prm; fun rm@prm = ··· ;
  let b1 = ref fls; // call to f active
  adv pf = λz.λx.let b′ = !b1; b1 := tru; let y = z x; b1 := b′; y;
  let b2 = ref fls; // call un occurred
  adv pun = λz.λx.b2 := tru; z x;
  (f, un, rm, λy.adv prm = λz.λx.if !b1 or !b2 then z x else y z x);
let (f,un,rm,pf_hsac) = Hsac ();
```

Advice attached using pf_hsac applies only in the specified conditions, and no other pointcuts are exposed. Again, the congruence results (c.f. Remark 12) apply equally to such terms. □

3.7 Access Control and Type Enforcement

We demonstrate how Type Enforcement (TE) [64, 65] policies—a form of history-sensitive mandatory access control popularized in the NSA's Security-Enhanced Linux (SELinux) [66]—can be encoded as temporal advice. TE policies associate types with code and other resources to be protected; henceforth, we call these "TE types" to avoid confusion with the usual notion of type found in programming languages. Also, the runtime system associates a current TE type with running code, which determines its privileges: access control decisions are based upon the current TE type and the TE type associated with the resource being accessed. The current TE type evolves as new code is invoked based upon (1) the current TE type, (2) the TE type associated with the new

code, (3) the TE policy, and (4) constraints imposed by the caller. The mechanism permits access control policies that are sensitive to the history of the code that has been executed and constraints imposed by that code.

Example 15 *(Web server).* As an example policy, consider a web server permitted to listen on ports 80 and 8080 if run by a system administrator, but only upon port 8080 if executed by an ordinary user. In this scenario, access privileges depend on both the original identity (system administrator or user) and the code (the web server) that is running.

To encode this policy, we allow the current TE type to range over $\{\mathsf{adm}, \mathsf{usr}, \mathsf{ws\_adm}, \mathsf{ws\_usr}\}$, the TE type for the web server code is ws_exe, and the TE types associated to the ports are $\{\mathsf{port}_{80}, \mathsf{port}_{8080}\}$. Initially, the current TE type is adm or usr, then when the web server is executed, the policy causes the current TE type to change from adm to ws_adm or from usr to ws_usr. In addition, the policy permits

- adm and usr to execute code of TE type ws_exe;
- ws_adm to access ports of TE type $\mathsf{port}_{80}, \mathsf{port}_{8080}$;
- ws_usr to access ports of TE type port_{8080}.

With this policy, the desired security property is a noninterference property, namely that the code running as usr cannot be influenced by new connections on port 80, even after executing other code. □

The TE mechanism can be implemented with advice, where protected resources are modeled as functions that can be advised. To define the advice, we require:

- A finite set of current TE types T and a finite set of TE types E for executable code, not necessarily disjoint.
- An "allow" relation $\mathit{allow} \subseteq T \times E \times T$ describes when code can execute/access a function and transition to a new TE type, i.e., if the current TE type is t then a function marked with TE code type e can be invoked successfully and transition to current TE type t' if $\mathit{allow}(t, e, t')$.
- An "automatic transition" map $\mathit{auto} : T \times E \to T$ describes TE type transitions that occur automatically when a new function is executed, i.e., if the current TE type is t and a function marked with TE code type e is invoked successfully, then it is executed with TE type $\mathit{auto}(t, e)$.
- A finite set of primitive pointcuts Q and a map $\mathit{type} : Q \to E$.

Although we do not do so here, it is straightforward to also incorporate nonautomatic transitions (c.f. the SELinux function setexeccon) that allow a caller to choose, subject to *allow*, a TE type other than the default "automatic" TE type.

We consider declarations $\mathsf{A}_{\mathsf{curr}}(t)$ representing a private variable curr storing the current TE type initialized with t. The scope of the private variable curr extends over advice A_q, one for each primitive pointcut q, which checks whether a call to a function at q is permitted and updates the current TE type before the call takes place. A free variable fail is invoked when an access control check fails. The coding for updating the current TE type uses the same strategy adopted for cflow in Example 13, i.e., the caller's current TE type is stored before proceeding and restored afterward.

$$\mathsf{A}_{\mathsf{curr}}(t) \triangleq \mathsf{pcd}\ p\,;\ \mathsf{fun}\ \mathsf{curr@}p = \lambda\,.\,t$$

$$
\begin{aligned}
\vec{\mathsf{A}}_Q &\triangleq (\mathsf{A}_q \mid q \in Q)\\
\mathsf{A}_q &\triangleq \mathsf{adv}\ q = \lambda z.\lambda x.\mathsf{L}_{z,x,q}\\
\mathsf{L}_{z,x,q} &\triangleq \mathsf{let\ next} = \mathit{auto}(\mathsf{!curr}, \mathit{type}(q));\\
&\quad \mathsf{if}\ \mathit{allow}(\mathsf{!curr}, \mathit{type}(q), \mathsf{next})\ \mathsf{then}\\
&\quad\quad \mathsf{let\ prev} = \mathsf{!curr};\ \mathsf{curr} := \mathsf{next};\\
&\quad\quad \mathsf{let}\ \mathsf{y} = z\ x;\ \mathsf{curr} := \mathsf{prev};\ \mathsf{y}\\
&\quad \mathsf{else}\quad \mathsf{fail}\ ()
\end{aligned}
$$

With the TE policy described in Example 15, and using TE code types as primitive pointcuts, suppose we are given a function webserver@ws_exe that starts a web server on a port given as an argument and functions $\mathsf{listen}_{80}@\mathsf{port}_{80}$, $\mathsf{listen}_{8080}@\mathsf{port}_{8080}$ that create listening sockets on ports 80 and 8080, respectively. We have:

$$
\begin{aligned}
T &\triangleq \{\mathsf{adm}, \mathsf{usr}, \mathsf{ws\_adm}, \mathsf{ws\_usr}, \mathsf{sys}, \mathsf{port}_{80}, \mathsf{port}_{8080}, \}\\
E &\triangleq \{\mathsf{ws\_exe}, \mathsf{port}_{80}, \mathsf{port}_{8080}\}
\end{aligned}
$$

With the *allow* relation specified by:

$$
\begin{array}{ll}
\mathit{allow}(\mathsf{adm}, \mathsf{ws\_exe}, \mathsf{ws\_adm}) & \mathit{allow}(\mathsf{usr}, \mathsf{ws\_exe}, \mathsf{ws\_usr})\\
\mathit{allow}(\mathsf{ws\_adm}, \mathsf{port}_{80}, \mathsf{sys}) & \\
\mathit{allow}(\mathsf{ws\_adm}, \mathsf{port}_{8080}, \mathsf{sys}) & \mathit{allow}(\mathsf{ws\_usr}, \mathsf{port}_{8080}, \mathsf{sys})
\end{array}
$$

Here the *auto* type transitions are exactly those allowed by *allow*. In the general type enforcement model, *auto* type transitions need not be the same as those allowed by *allow*, because *allow* can include nonautomatic transitions.

Now, in the absence of $\mathit{allow}(\mathsf{ws\_usr}, \mathsf{port}_{80}, \mathsf{sys})$, the advice implementing the TE policy prevents the web server from accessing port 80 when invoked with a current TE type of usr, i.e., if webserver attempts to invoke listen_{80} in the following program, the advice implementing the TE policy will cause fail to be invoked instead, because the invocation of webserver will cause the current TE type to change to ws_usr.

$$\mathsf{A}_{\mathsf{curr}}(\mathsf{usr});\ \vec{\mathsf{A}}_Q;\ \mathsf{webserver}\ (80)$$

In this example, we see that the body of $\mathsf{listen}_{80}@\mathsf{port}_{80}$ is irrelevant to computation beginning with TE type usr. To formalize this noninterference property, we first define reachability $\mathit{reach}(t, e)$ of a TE type e from a TE type t to be the least relation such that:

- $\exists t'.\ \mathit{allow}(t, e, t')$ implies $\mathit{reach}(t, e)$
- $\exists t', e'.\ \mathit{allow}(t, e', t')$ and $\mathit{reach}(t', e)$ implies $\mathit{reach}(t, e)$

Reachability $\mathit{reach}(t, q)$ of a primitive pointcut from a TE type t is then defined to hold exactly when $\mathit{reach}(t, \mathit{type}(q))$. In the example above, the TE code type port_{80} is not reachable from usr.

The desired noninterference property is that we can take a program that declares functions at public primitive pointcuts, impose advice for type enforcement on those public primitive pointcuts, then arbitarily change the bodies of functions declared at primitive pointcuts unreachable from the initial TE type without changing the behavior of the program. In this already long paper, we elide the treatment and formal proof of this property for our encoding.

4 Labeled Transition System and Bisimulation

In this section, we present the bisimilarity relation following the LTS style pioneered by Gordon [14, 43], in particular in the style of presentation of Jeffrey and Rathke for Concurrent ML [17]. In contrast to this prior work, our intuitions are guided by open bisimulation and address aspect features. The technical consequence of this difference is that our proof that bisimilarity is a congruence is a direct proof based on a direct analysis of substitutions rather than following these papers in being based on Sangiorgi [67] or Howe [44].

The rest of this section is organized as follows. In Section 4.1, we describe the ideas of our LTS for the restricted case of the pure untyped lambda calculus without aspects or declarations. This treatment of a familiar calculus is intended to motivate the use of symbolic functions and advice in the LTS and to provide core intuitions for the following subsections. In Sect. 4.2, we adapt the operational semantics of earlier sections to deal with symbolic data such as functions and advice. In Sect. 4.3, we provide a description of the LTS for the full calculus, and follow with a definition of the bisimilarity relation in Sect. 4.4. Section 4.5 makes the intuitions of our model concrete by a series of examples.

4.1 An Introduction to Open Bisimulation

In this Subsection, we provide an snapshot of our approach by briefly describing an LTS for the pure untyped call-by-value lambda calculus.

We briefly recall the LTS approach [43] to applicative bisimulation for the pure untyped call-by-value lambda calculus

- A nonvalue term M has a τ transition to M' if M reduces in one step to M'.
- A value U (e.g., $\lambda x.M$) has a transition labeled U' to the application $U\ U'$.

Two terms are bisimilar if the associated transition systems are bisimilar, i.e., reduction of one term terminates iff reduction of the other term terminates, and, if the terms reduce to values, then the results of applying both values to the same value are again bisimilar.

Our approach is inspired by open bisimulation [48] and ENF-bisimulation [49, 50]. (The reader can view this subsection, in isolation, as a presentation of ENF-bisimulation-upto-η using an LTS.) Following our conventions, we use ϕ and ψ for variables that occur free in terms.

- A nonvalue term M has a τ transition to M' if M reduces in one step to M'.
- Values U have transitions labeled $\mathsf{app}\ \phi$ (where ϕ is fresh) to the application $U\ \phi$.
- Terms can now be of the form $\mathscr{E}[\phi\ U]$, for some evaluation context $\mathscr{E}$, where ϕ is an uninterpreted symbol. These terms have additional transitions (similar to ENF-bisimulation [49, 50]):
  - A transition labeled $\mathsf{fcall}\ \phi$ to U.
  - Transitions labeled $\mathsf{ret}\ \psi$ to $\mathscr{E}[\psi]$ for a fresh environment variable ψ.

Again, two terms are bisimilar if the associated transition systems are bisimilar.

The fcall ϕ transitions ensure that if the application $\mathscr{E}[\phi\ U]$ is bisimilar to the application $\mathscr{E}'[\phi'\ U']$ then $\phi = \phi'$ and U is bisimilar to U'. Similarly, the ret ψ transitions ensure that if $\mathscr{E}[\phi\ U]$ is bisimilar to $\mathscr{E}'[\phi'\ U']$ then $\mathscr{E}[\psi]$ is bisimilar to $\mathscr{E}'[\psi]$.

We extend this approach to open bisimulation by adding the features required to accommodate state and symbolic advice.

4.2 Symbolic Functions and Symbolic Advice

The LTS must allow functions and advice to be defined by the environment, influencing a term. To accommodate context functions, we need only to extend our notion of wellformedness to allow occurrences of free variables representing these functions. As noted earlier, we use ϕ, ψ to indicate these free variables; we sometimes refer to these as *symbolic function names* because they are uninterpreted in the term.

To accommodate context advice, we assume a countably infinite set of *symbolic advice names*, α, β, disjoint from the sets of variable names and primitive pointcuts.

SYMBOLIC ADVICE

α, β	Symbolic Advice Names
$A, B ::= \cdots \mid \mathsf{adv}\ p = \alpha$	Symbolic Advice Declaration
$U, V, W ::= \cdots \mid \alpha\langle U \rangle$	Symbolic Advice Call

A symbolic advice call $\alpha\langle U \rangle$ binds the proceed variable for α to U. Symbolic advice call typically occurs with a further application to the argument of the function being advised. Thus, $\alpha\langle U \rangle\ V$ indicates that the context is executing advice α with proceed value U and argument V.

Note that if $A = \mathsf{fun}\ f@p = \phi$, then by our previous definition of lookup $\vec{A}(f) = \langle p, \phi \rangle$; thus no extensions are required to handle symbolic functions. For symbolic advice, we extend the definition of *advise* as follows.

$$advise(q, U, \mathsf{adv}\ p = \alpha\,;\vec{A}) \triangleq \begin{cases} advise(q, \alpha\langle U \rangle, \vec{A}) & \text{if } p = q \\ advise(q, U, \vec{A}) & \text{otherwise} \end{cases}$$

Example 16 *(Evaluation with symbolic names).* Let

$$\vec{\mathsf{A}} = \mathbf{pcd}\ \mathsf{p}\,;\ \mathbf{fun}\ \mathsf{f@p} = \phi\,;\ \mathbf{adv}\ \mathsf{p} = \alpha\,;\ \mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\lambda \mathsf{x}.(\mathsf{z}\ \mathsf{x}) * 3.$$

Evaluation of $\mathsf{f}\,()$ proceeds as follows.

$$\begin{aligned} \vec{\mathsf{A}}/\mathsf{f}\,() &\longrightarrow\!\!\!\!\!\rightarrow \vec{\mathsf{A}}/(\lambda \mathsf{x}.(\alpha\langle \phi \rangle\ \mathsf{x}) * 3)\,() \\ &\longrightarrow\!\!\!\!\!\rightarrow \vec{\mathsf{A}}/(\alpha\langle \phi \rangle\,()) * 3 \end{aligned}$$

Evaluation is now stuck; intuitively, control is given to the context that defined α.

Note that if evaluation arrives at an application $\mathsf{f}\,()$, then the result is ϕ; again evaluation is stuck, this time giving the context control through the undefined body of ϕ. □

The following special form of substitution is used in proofs.

Definition 17 Substitution of an abstraction for symbolic advice "$M[\alpha := \lambda z.U]$" is defined homorphically over terms, with the only interesting case being for calls to symbolic advice:

$$(\alpha\texttt{<}V\texttt{>})[\alpha := \lambda z.U] \triangleq U[z := (V[\alpha := \lambda z.U])] \qquad \square$$

4.3 The LTS

We now define the states of the LTS. For namespace management, these include a *symbol environment*, which binds all symbolic function and advice names, and a *symbol declaration*, which may declare primitive pointcuts, functions, and advice.

LTS SYNTAX

$\mathbf{M},\mathbf{N} ::= \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$	Configuration
$\Gamma ::= \cdot \mid \phi,\Gamma \mid \alpha,\Gamma$	Symbol Environment
$\Delta ::= \cdot \mid A,\Delta$	Symbol Declaration

In a configuration $\vec{A}/\vec{\mathscr{E}}/M/\vec{U}$, we refer to M as the *active term*.

With respect to evaluation configurations, the new elements are the list of contexts $\vec{\mathscr{E}}$ and the list of values $\vec{U}$. The contexts $\vec{\mathscr{E}}$ model the call stack: it will be used in a manner consistent with the stack discipline. The list $\vec{U}$ includes all values that have been released/leaked to the environment during evaluation of the term. Thus, the values in $\vec{U}$ are available for the environment to inspect and use. Formally, $\vec{U}$ is a way to account for the imperative/state features of the calculus. These modeling ideas follow prior research [17, 18, 61].

We define the LTS relative to a *symbol environment* Γ and *symbol declaration* Δ. In Sect. 4.4, we will define bisimilarity as $\Gamma;\Delta \vdash \mathbf{M} \sim \mathbf{N}$. The symbol environment is used to manage names in the LTS, in particular to ensure two bisimilar terms may always make transitions with the same labels. The symbol declaration, likewise, ensures that both contexts in a bisimulation have the same observation power. (We describe how to derive an initial state from a term in Definition 20.)

The target symbol environment/declaration of a transition is determined by the source symbol environment/declaration and the label of the transition.

Definition 18 *(LTS state).* In a configuration $\vec{A}/\vec{\mathscr{E}}/M/\vec{U}$, $dn(\vec{A})$ are bound in $\vec{\mathscr{E}}/M/\vec{U}$. (The let binders in $\vec{\mathscr{E}}$ are not in scope in M or $\vec{U}$ and thus are not binding.)

A *state* of the LTS is a triple $\Gamma;\Delta \vdash \mathbf{M}$, where the names listed in Γ are bound in Δ and $\mathbf{M}$ and $dn(\Delta)$ are bound in $\mathbf{M}$. A state is *well formed* if no name occurs free, and no name is declared more than once in $\Gamma,\Delta,\vec{A}$. $\square$

By way of contrast with evaluation configurations, note that we require a well formed LTS state to be closed. In the sequel, we assume that all LTS states are well formed.

Names introduced by the context appear on the left side of the turnstile. Advice is unnamed, and thus advice introduced by the context appears on the right side of the turnstile in $\vec{A}$. Declarations introduced dynamically by the context appear on the right side of the turnstile in $\vec{A}$. Thus, $\vec{A}$ will include symbolic advice laid down by the

environment—so, in general, it can be an interleaving of definitions of advice placed by the term and by the context.

We now present the labels used in the LTS.

LTS LABELS

$\varkappa ::= \tau \mid \kappa$	All Labels
$\kappa ::=$	Visible Labels
fcall ϕ	Term calls context function ϕ
acall α	Term calls context advice α
ret ϕ	Context returns to term with result ϕ ($dn = \{\phi\}$)
app ϕ	Context calls term with argument ϕ ($dn = \{\phi\}$)
put	Context saves value
get i	Context restores value
fun f@$p = \phi$	Context declares function ($dn = \{f, \phi\}$)
adv $p = \alpha$	Context declares advice ($dn = \{\alpha\}$)

In Gordon's terminology [43], the visible labels fcall and acall are *active* (representing actions initiated by the term), whereas the remaining visible labels are *passive* (representing actions initiated by the environment). The choice of labels is determined by the possibilities available to the context to interact with the term. The examples in this section show how the labels correspond precisely to such contexts—an informal argument that underlies the completeness theorem (Appendix E).

LTS

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \xrightarrow{\tau} \Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{E}}/N/\vec{U} \quad \text{if } \Delta,\vec{A}/M \longrightarrow \Delta,\vec{B}/N$$

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{F}[\phi\ V]/\vec{U} \xrightarrow{\text{fcall } \phi} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/V/\vec{U}$$

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{F}[\alpha\langle V\rangle\ W]/\vec{U} \xrightarrow{\text{acall } \alpha} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/W/\vec{U},V$$

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/V/\vec{U} \xrightarrow{\text{ret } \phi} \Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{F}[\phi]/\vec{U}$$

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U} \xrightarrow{\text{app } \phi} \Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V\ \phi/\vec{U}$$

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U} \xrightarrow{\text{put}} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U},V$$

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U} \xrightarrow{\text{get } i} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U_i/\vec{U} \quad \text{if } 1 \le i \le |\vec{U}|$$

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U} \xrightarrow{\text{fun } f@p=\phi} \Gamma,\phi;\Delta,\text{fun } f@p = \phi \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U},f \quad \text{if } p \in dn(\Delta)$$

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U} \xrightarrow{\text{adv } p=\alpha} \Gamma,\alpha;\Delta \vdash \vec{A}\,;\,\text{adv } p = \alpha/\vec{\mathscr{E}}/V/\vec{U} \quad \text{if } p \in dn(\Delta)$$

The fact that configurations must be well formed ensures that, in the rules for ret and app, the name ϕ must be fresh (i.e., must not occur in $\Gamma \cup dn(\Delta) \cup dn(\vec{A})$; likewise for the names ϕ and f in the rule for fun and α in the rule for adv. Wellformedness also ensures that in the rules for fcall and acall, ϕ and α must occur in Γ.

The LTS has several significant properties:

Call-by-value invariant. The LTS rules enforce a call-by-value invariant. This is seen by noting that precedence is afforded to internal reductions of the term. So, all rules except the first three are applicable to a state $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ only if M is a value.

Applicative tests. app ϕ performs applicative tests. Rather than providing a term as an argument for the applicative test, this rule provides a fresh symbolic argument ϕ.

Stack of evaluation contexts. In the pure lambda calculus setting of Section 4.1, the rules for fcall and ret reflect the absence of interference between the caller and the callee in a purely functional language—the testing of the evaluation context and the callee argument is done separately. Thus, there was no need to track the evaluation context in the LTS for the pure lambda calculus.

In contrast, the LTS for the full calculus has to permit the environment an opportunity to inspect the arguments before the term continues evaluation—this is meaningful for the full calculus because of state changes caused by the dynamic laying down of advice. This is done in our LTS by the use of the stack of evaluation contexts $\mathscr{E}$.

fcall ϕ pushes the current evaluation context into $\mathscr{E}$. The active term becomes the argument to the call V. The transition ret ϕ returns a symbolic value ϕ to the top evaluation frame, $\mathscr{F}$, of the stack $\vec{\mathscr{E}}, \mathscr{F}$ and moves it into the current-term position, popping $\mathscr{F}$ from the top of the stack. (This stack discipline would have to be liberalized to address a language with control operators.)

Note that calls to signal (from Section 3.4) are treated like any other call, and thus generate labels of the form fcall signal.

Symbolic advice tests. In the rule for acall, the argument V is added to the list of values that are available for the environment to inspect and use. As in the case for fcall, the active term is changed to the argument, in this case W.

Environment value tests. put and get enable the movement of values between $\vec{U}$, the list of values leaked to the environment, and the active position of the configuration. put permits an evaluated value to be saved for use by the environment. get permits the environment to interact with a saved argument by moving it into the active term position. This rule leaves a copy of the restored term in $\vec{U}$. The label on this rule carries the position i in $\vec{U}$ that is being restored. Conceptually, put and get ensure that $\vec{U}$ is closed under structural rules.

New name tests. The rules for fun and adv permit the environment to add new function names and new advice. The first rule is necessary for bookkeeping; it allows the context to create an unbounded number of new function names; new names are added to the list of values $\vec{U}$ to maintain the invariant that functions declared in Δ can be inspected by the environment. The second rule is needed for more than bookkeeping. Since the order of advice matters, the rule for adv $p = \alpha$ also has to insert it into the list of advice declarations being carried in $\vec{A}$.

4.4 Bisimulation

Define $\twoheadrightarrow$ to be the reflexive transitive closure of $\xrightarrow{\tau}$. On visible labels define the weak-labeled transition relation $\overset{\kappa}{\twoheadrightarrow}$ as $\twoheadrightarrow \xrightarrow{\kappa} \twoheadrightarrow$.

Note in the definition of the LTS $(\Gamma; \Delta \vdash \mathbf{M} \xrightarrow{\varkappa} \Gamma'; \Delta' \vdash \mathbf{M}')$ that the symbol environment and declaration in the residual $(\Gamma'; \Delta')$ are uniquely determined by the initial state $(\Gamma; \Delta)$ and label $(\varkappa)$. This leads us to define bisimilarity as a family of relations between configurations, written $\Gamma; \Delta \vdash \mathbf{M} \sim \mathbf{N}$. It is technically convenient to require that bisimilar configurations have equal length lists of contexts and values. (Alternatively,

we could prove that these invariants hold for bisimulations derived from the initial configurations of Definition 20.)

Definition 19. We say that a configuration $\vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ has sort $\langle \Gamma, \Delta, m, n\rangle$ if $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ is well formed, the length of $\vec{\mathscr{E}}$ is m, and the length of $\vec{U}$ is n.

We define similarity, $\lesssim$, as the largest family of $\langle \Gamma, \Delta, m, n\rangle$-indexed relations over configurations such that

$$\Gamma;\Delta \vdash \mathbf{M} \lesssim \mathbf{N} \text{ and } \Gamma;\Delta \vdash \mathbf{M} \xrightarrow{\kappa} \Gamma';\Delta' \vdash \mathbf{M}'$$

imply that for some $\mathbf{N}'$

$$\Gamma;\Delta \vdash \mathbf{N} \xrightarrow{\kappa} \Gamma';\Delta' \vdash \mathbf{N}' \text{ and } \Gamma';\Delta' \vdash \mathbf{M}' \lesssim \mathbf{N}'.$$

$\Gamma;\Delta$-*bisimilarity*, $\sim$ is defined as two way similarity:

$$\Gamma;\Delta \vdash \mathbf{M} \sim \mathbf{N} \text{ if } \Gamma;\Delta \vdash \mathbf{M} \lesssim \mathbf{N} \text{ and } \Gamma;\Delta \vdash \mathbf{N} \lesssim \mathbf{M}. \qquad \square$$

For a fixed sort, $\langle \Gamma, \Delta, m, n\rangle$-indexed relations over configurations form a lattice ordered by set inclusion. The product of all these lattices over all sorts yields a lattice structure on sort-indexed families of relations. Indexed (bi)similarity can also be formalized as the greatest fixed point of a monotone operator on this lattice. As is standard, Definition 19 describes $\lesssim$ (resp. $\sim$) as the prefixed point of this monotone operator.

Relational composition of two sort-indexed families of relations is defined by pointwise composition. For example, let $\mathscr{R} = \{\mathscr{R}_{\langle \Gamma,\Delta,m,n\rangle}\}$ and $\mathscr{S} = \{\mathscr{S}_{\langle \Gamma,\Delta,m,n\rangle}\}$ then $\mathscr{R} \circ \mathscr{S}$ is the sort-indexed family $\{\mathscr{R}_{\langle \Gamma,\Delta,m,n\rangle} \circ \mathscr{S}_{\langle \Gamma,\Delta,m,n\rangle}\}$. Since the tests of open bisimilarity are only names, standard proofs for first order calculi [68] apply here to yield that $\lesssim$ (resp. $\sim$) is closed under composition. Thus, $\lesssim$ (resp. $\sim$) is a preorder (resp. an equivalence relation).

Bisimilarity is insensitive to the addition of irrelevant new names to Γ, i.e., if $\Gamma;\Delta \vdash \mathbf{M} \sim \mathbf{N}$ and $\Gamma' \cap \Gamma = \emptyset$, then $\Gamma, \Gamma';\Delta \vdash \mathbf{M} \sim \mathbf{N}$. Symmetrically, bisimilarity is also insensitive to the removal of irrelevant new names from Γ, i.e., names in Γ that are not free in the rest of the configuration can be removed. The proofs follow by noting that the names in Γ are only typing constraints; so, addition or removal of names does not alter the transition capabilities of a configuration.

Bisimilarity on configurations relates to terms as follows.

Definition 20. Write "$\Gamma;\Delta \vdash M \sim N$" if $\Gamma;\Delta \vdash (\cdot/\cdot/M/\vec{f}) \sim (\cdot/\cdot/N/\vec{f})$, where $\vec{f}$ are the function names bound in Δ, in declaration order.

Write "$M \sim N$" if $(\vec{\phi}, \vec{\alpha});(\mathsf{pcd}\ \vec{p}\,;\ \mathsf{adv}\ \vec{p} = \vec{\alpha}) \vdash M \sim N$, where $fn(M,N) = \{\vec{\phi}, \vec{p}\}$. $\square$

The function symbols $\vec{\phi}$ detect function calls by the term. The primitive pointcut declarations $\mathsf{pcd}\ \vec{p}$ bind the free primitive pointcuts in the term. The advice declarations $\mathsf{adv}\ \vec{p} = \vec{\alpha}$ detect any call to a new function declared at a visible primitive pointcut (by the term). Functions can be introduced by fun transitions to detect any new advice declared at a visible primitive pointcut (by the term).

4.5 Simple Examples

The first examples show that bisimilarity yields a β_v, η_v theory, i.e., that the call-by-value versions of β- and η-equality preserve bisimilarity.

Example 21 *(β_v preserves bisimilarity).* A standard LTS proof shows that prefixing by τ preserves bisimilarity. So, since:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/(\lambda x.M)\ U/\vec{U} \xrightarrow{\tau} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M[x := U]/\vec{U},$$

and there are no other transitions to consider, we have

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/(\lambda x.M)\ U/\vec{U} \sim \vec{A}/\vec{\mathscr{E}}/M[x := U]/\vec{U}.$$

Thus, β_v preserves bisimilarity. □

Example 22 *(η_v preserves bisimilarity).* η_v holds, i.e., in the case where x is not free in U, we have

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{A}/\vec{\mathscr{E}}/\lambda x.Ux/\vec{U}.$$

The key case in establishing the above bisimulation is to note that the transition

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \xrightarrow{\mathsf{app}\,\phi} \Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U\ \phi/\vec{U}$$

on the LHS is matched by the following sequence from the RHS, starting

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\lambda x.Ux/\vec{U} \xrightarrow{\mathsf{app}\,\phi} \Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/(\lambda x.Ux)\ \phi/\vec{U}$$

and continuing

$$\Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/(\lambda x.Ux)\ \phi/\vec{U} \xrightarrow{\tau} \Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U\ \phi/\vec{U}.$$ □

The other cases are addressed by a simple bisimulation relation that relates configurations that are identical except that the occurrences of U in the active term and the value list can be replaced by $(\lambda x.Ux)$.

Bisimilarity is not a trivial relation: for example, it distinguishes the Church booleans from one another, and likewise the Church numerals.

Example 23 *(Bisimulation is sensitive to advice order).* We revisit Example 9, noting that M and M' are distinguished by bisimilarity, where

$$\mathsf{M} = \mathbf{fun}\ \mathsf{f@p} = \lambda \mathsf{x}.0;\ \mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U}_1;\ (\lambda.(\mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U}_2;\mathsf{f}\ 0))$$
$$\mathsf{M}' = \mathbf{fun}\ \mathsf{f@p} = \lambda \mathsf{x}.0;\ \mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U}'_1;\ (\lambda.(\mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U}'_2;\mathsf{f}\ 0))$$

and

$$\mathsf{U}_1 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+0) \qquad \mathsf{U}'_1 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+1)$$
$$\mathsf{U}_2 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+1) \qquad \mathsf{U}'_2 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+0).$$

The term M can perform the following transitions in the LTS, whereas M' can make all transitions but the last one. The transitions correspond to the context presented in Example 9.

τ	*Term places advice* U_1 *and returns* $(\lambda.(\mathbf{adv}\ \mathsf{p} = \lambda\mathsf{z}.\mathsf{U}_2\,;\mathsf{f}\ 0))$
$\mathsf{adv}\ \mathsf{p} = \alpha$	*Context defines advice*
$\mathsf{app}\ \phi$	*Context calls* $(\lambda.(\mathbf{adv}\ \mathsf{p} = \lambda\mathsf{z}.\mathsf{U}_2\,;\mathsf{f}\ 0))$ *with ignored argument* ϕ
τ^*	*Term places advice* U_2 *and calls* f *with argument* 0
$\mathsf{acall}\ \alpha$	*Term calls* α *with argument with argument* 1
$\mathsf{app}\ \psi_{\mathsf{succ}}$	*Context provides first argument to Church encoding of* 1
$\mathsf{app}\ \psi_{\mathsf{zero}}$	*Context provides second argument to Church encoding of* 1
$\mathsf{fcall}\ \psi_{\mathsf{succ}}$	*Church encoding of* 1 *identifies itself by calling back to* ψ_{succ} □

As demonstrated in Example 21, the order of evaluation and multiplicity of use of "internal" functions are not necessarily detectable. Bisimilarity can, however, detect the order and multiplicity of calls to symbolic functions created by the environment. This corresponds to the opponent (the context) having state.

Example 24 *(Detecting order).* Consider the following terms.

$$\mathbf{let}\ \mathsf{x} = \phi\,()\,;\mathbf{let}\ \mathsf{y} = \psi\,()\,;\,()$$
$$\mathbf{let}\ \mathsf{y} = \psi\,()\,;\mathbf{let}\ \mathsf{x} = \phi\,()\,;\,()$$

The LTSs for these terms are immediately distinguished by the initially enabled transition, namely $\mathsf{fcall}\ \phi$ for the first term and $\mathsf{fcall}\ \psi$ for the second. □

Example 25 *(Detecting multiplicity).* Consider the following terms.

$$\mathbf{let}\ \mathsf{x} = \phi\,()\,;\mathbf{let}\ \mathsf{y} = \phi\,()\,;\,()$$
$$\mathbf{let}\ \mathsf{y} = \phi\,()\,;\,()$$

The LTSs for the first term may perform the following sequence of transitions: $\mathsf{fcall}\ \phi$, $\mathsf{ret}\ \psi$, and $\mathsf{fcall}\ \phi$. The second term can match the first two of these transitions, but not the third. □

The distinctions made in Examples 24 and 25 (which are necessary in the full language with imperative features) hold even if all of the terms involved are purely functional, i.e., have no aspects.

Example 26 *(The use of get and put rules).* Consider the following terms.

$$\mathsf{M} = \vec{\mathsf{A}}\,;\mathsf{U} \qquad \vec{\mathsf{A}} = \mathbf{pcd}\ \mathsf{p}\,;\mathbf{fun}\ \mathsf{f@p} = \lambda.\mathsf{fls}\,;$$
$$\mathsf{V} = \lambda.\mathsf{tru} \qquad \mathsf{U} = \lambda.\mathbf{let}\ \mathsf{x} = \mathbf{not}(\mathsf{f}\ ())\,;(\mathbf{adv}\ \mathsf{p} = \lambda.\lambda.\mathsf{x})\,;\mathsf{x}$$

V is a function that always returns tru, whereas U (the function returned by M) will alternately return tru and fls, because of the state changes caused by the aspect in U. The terms can be distinguished by the context

$$\mathscr{E} = \mathbf{let}\ \mathsf{y} = [-]\,;\mathsf{y}\,()\,;\mathsf{y}\,()$$

since $\mathscr{E}[\mathsf{V}]$ yields tru and $\mathscr{E}[\mathsf{M}]$ yields fls.

Clearly, this distinction relies crucially on the use of U twice. In the following example, we essentially show that the LTS is expressive enough to code the distinguishing context $\mathscr{E}$ by using put, get tests to permit multiple tests of terms.

Using the definitions above, the behavior of $\mathscr{E}$ can be simulated in the LTS using the put, get rules as follows. Consider the initial configuration $\cdot;\cdot \vdash \cdot/\cdot/\mathsf{M}/\cdot$, which has τ transitions to $\cdot;\cdot \vdash \vec{\mathsf{A}}/\cdot/\mathsf{U}/\cdot$. This configuration in turn has a put labeled transition to

$$\cdot;\cdot \vdash \vec{\mathsf{A}}/\cdot/\mathsf{U}/\mathsf{U},$$

which in turn has an app ϕ labeled transition to

$$\phi;\cdot \vdash \vec{\mathsf{A}}/\cdot/\mathsf{U}\ \phi/\mathsf{U}.$$

A few τ transitions from this configuration yields

$$\phi;\cdot \vdash \vec{\mathsf{A}}\,;\, \mathbf{adv}\,\mathsf{f} = \lambda.\lambda.\mathsf{tru}/\cdot/\mathsf{tru}/\mathsf{U}.$$

To reevaluate U, we use a get 1 transition to get

$$\phi;\cdot \vdash \vec{\mathsf{A}}\,;\, \mathbf{adv}\,\mathsf{f} = \lambda.\lambda.\mathsf{tru}/\cdot/\mathsf{U}/\mathsf{U}.$$

An app ψ labeled transition yields

$$\phi,\psi;\cdot \vdash \vec{\mathsf{A}}\,;\, \mathbf{adv}\,\mathsf{f} = \lambda.\lambda.\mathsf{tru}/\cdot/\mathsf{U}\ \psi/\mathsf{U}.$$

This second evaluation of U takes place in the context of the aspect that has been laid down. A few τ transitions from this configuration yields

$$\phi,\psi;\cdot \vdash \vec{\mathsf{A}}\,;\, \mathbf{adv}\,\mathsf{f} = \lambda.\lambda.\mathsf{tru}\,;\, \mathbf{adv}\,\mathsf{f} = \lambda.\lambda.\mathsf{fls}/\cdot/\mathsf{fls}/\mathsf{U}. \qquad \square$$

Much of the related work is formalized in terms of references, rather than advisable functions. In the next example, we discuss some of the subtleties, using the work of Meyer and Sieber [21] as the basis for comparison.

Example 27 *(Primitive references versus advisable functions).* For a free reference variable x, Meyer-Sieber [21] validate the equivalence $!\mathsf{x};\, !\mathsf{x} \overset{\text{MS}}{=} !\mathsf{x}$. In our encoding of references, this translates roughly to the *in*equivalence demonstrated in Example 25. The difference arises from the weak assumptions one can make about functions relative to references; indeed the equivalence is valid in our language for *bound* references, where stronger assumptions are manifest:

$$\mathbf{let}\,\mathsf{x} = \mathsf{ref}\ 0\,;\, !\mathsf{x};\, !\mathsf{x} \sim \mathbf{let}\,\mathsf{x} = \mathsf{ref}\ 0\,;\, !\mathsf{x}.$$

Unwinding the definition of references, this is roughly

$$\mathbf{pcd}\,\mathsf{p}\,;\,\mathbf{fun}\,\mathsf{f@p} = \lambda.0\,;\,\mathsf{f}\,()\,;\,\mathsf{f}\,() \sim \mathbf{pcd}\,\mathsf{p}\,;\,\mathbf{fun}\,\mathsf{f@p} = \lambda.0\,;\,\mathsf{f}\,().$$

But the equivalence does not hold when p is available to the context, since calls to f are then observable. Let $\Delta = \mathbf{pcd}\,\mathsf{p}, \mathbf{fun}\,\mathsf{f@p} = \lambda.0$. Then

$$\cdot;\Delta \vdash \mathsf{f}\,()\,;\,\mathsf{f}\,() \not\sim \mathsf{f}\,().$$

Interestingly, the equivalence does hold after an assignment, i.e., declaration of nonproceeding advice. Let $\mathsf{A} = \mathbf{adv}\ \mathsf{p} = \lambda.\lambda.1$, then

$$\cdot;\Delta \vdash \mathsf{A};\mathsf{f}();\mathsf{f}() \sim \mathsf{A};\mathsf{f}()$$

which corresponds to $(\mathsf{x}:=1;!\mathsf{x};!\mathsf{x}) \overset{\text{MS}}{=} (\mathsf{x}:=1;!\mathsf{x})$.

Note also that for pure references $!\mathsf{x};\Omega \overset{\text{MS}}{=} \Omega$, whereas the corresponding result for functions does not hold: $\mathsf{f}();\Omega \overset{\text{MS}}{\neq} \Omega$. □

4.6 A Reasoning Principle

To simplify reasoning about bisimilarity, we develop an upto principle that eliminates the need to:

- Replicate values in bisimulations, e.g., arising from a get 1 then a put transition.
- Include terms that do not interact with the state if they occur in the same position on each side of the bisimulation.

The following definition formalizes the above notion of replicated values.

Definition 28. $\mathscr{R}^{\bullet}_{\mathtt{dup}}$ is the least relation such that $\mathscr{R} \subseteq \mathscr{R}^{\bullet}_{\mathtt{dup}}$ and if

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/U,\vec{U}\ \ \mathscr{R}^{\bullet}_{\mathtt{dup}}\ \ \vec{B}/\vec{\mathscr{F}}/N/V,\vec{V}$$

then

$$\Gamma,\Gamma';\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/U,\vec{U},U\ \ \mathscr{R}^{\bullet}_{\mathtt{dup}}\ \ \vec{B}/\vec{\mathscr{F}}/N/V,\vec{V},V, \text{ and}$$
$$\Gamma,\Gamma';\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}\ \ \mathscr{R}^{\bullet}_{\mathtt{dup}}\ \ \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$$ □

We say that a term (resp. evaluation context) is *state free* over a symbol environment $(\Gamma;\Delta)$ if every free name is contained in Γ and the term (resp. evaluation context) contains *no* declaration subterms. The following definition formalizes the addition of identical state-free evaluation contexts (or values) to a relation on configurations.

Definition 29. $\mathscr{R}^{\bullet}_{\mathtt{sf}}$ is the least relation such that $\mathscr{R} \subseteq \mathscr{R}^{\bullet}_{\mathtt{sf}}$ and, for L (resp. W, $\mathscr{E}$) a state-free term (resp. value, context) for $(\Gamma,\Gamma';\Delta)$, we have that if

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}\ \ \mathscr{R}^{\bullet}_{\mathtt{sf}}\ \ \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$$

then the following hold.

$$\Gamma,\Gamma';\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/W,\vec{U}\ \ \mathscr{R}^{\bullet}_{\mathtt{sf}}\ \ \vec{B}/\vec{\mathscr{F}}/N/W,\vec{V}$$
$$\Gamma,\Gamma';\Delta \vdash \vec{A}/\mathscr{E},\vec{\mathscr{E}}/M/\vec{U}\ \ \mathscr{R}^{\bullet}_{\mathtt{sf}}\ \ \vec{B}/\mathscr{E},\vec{\mathscr{F}}/N/\vec{V}$$ □

Moreover, if M and N are values:

$$\Gamma,\Gamma';\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L/\vec{U}\ \ \mathscr{R}^{\bullet}_{\mathtt{sf}}\ \ \vec{B}/\vec{\mathscr{F}}/L/\vec{V}$$

Let $\mathscr{R}^{\bullet} = \mathscr{R}^{\bullet}_{\mathtt{dup}} \cup \mathscr{R}^{\bullet}_{\mathtt{sf}}$. Let $\nleftrightarrow$ be the reflexive, transitive closure of the least symmetric relation containing $\xrightarrow{\tau}$. The following upto-technique is used to prove equivalences in Sect. 4.7.

Lemma 30. *Let $\mathscr{R}$ be a $\langle \Gamma, \Delta, m, n\rangle$-indexed relation on configurations. Suppose that*

$$\Gamma;\Delta \vdash \mathbf{M} \;\; \mathscr{R} \;\; \mathbf{N} \text{ and } \Gamma;\Delta \vdash \mathbf{M} \xrightarrow{\kappa} \Gamma';\Delta' \vdash \mathbf{M}'$$

implies there exists $\mathbf{N}'$ such that

$$\Gamma;\Delta \vdash \mathbf{N} \xrightarrow{\kappa} \Gamma';\Delta' \vdash \mathbf{N}' \text{ and } \Gamma';\Delta' \vdash \mathbf{M}' \;\; (\twoheadleftarrow;\mathscr{R}^{\bullet};\twoheadleftarrow) \;\; \mathbf{N}'.$$

Then $(\twoheadleftarrow;\mathscr{R}^{\bullet};\twoheadleftarrow) \subseteq \sim$.

Proof Sketch. The complex statement of the lemma masks its simple (but lengthy) proof. As we have discussed already in Example 21, a simple and standard LTS proof shows that prefixing by τ preserves bisimilarity. So, $(\twoheadleftarrow;\sim;\twoheadleftarrow) \subseteq \sim$. Thus, the essence of the above lemma is that $\mathscr{R}^{\bullet} \subseteq \sim$. The proof formalizes the following intuitive observations:

- Transitions from duplicated values are already available in the starting configuration. So, duplicating values in the value list does not alter bisimilarity reasoning.
- State-free terms are "functional" in the sense that transitions from state-free terms are not dependent on the configuration. So, addition of identical state-free terms to the value list does not alter bisimilarity reasoning. □

One very useful consequence of the lemma is that $\sim^{\bullet} \subseteq \sim$. The results of Sect. 5 imply that $\sim$ is sound for a more general version of Definition 29: i.e., if $fn(U)$ and $fn(\mathscr{E})$ are bound by Γ and Δ and if $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \;\sim\; \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ then,

$$\Gamma;\Delta \vdash \vec{A}/\mathscr{E},\vec{\mathscr{E}}/M/U,\vec{U} \;\sim\; \vec{B}/\mathscr{E},\vec{\mathscr{F}}/N/U,\vec{V}.$$

However, this more general property of $\sim$ is not necessarily sound as part of an upto-proof technique.

4.7 Examples with Local Store and Higher-Order Functions

Examples 31 and 32 illustrate equivalences involving local state and higher-order functions—originally due to Meyer and Sieber [21]. The proofs provided here exemplify the techniques needed to address Examples 1–5 and Example 7 from [21]. Their Example 6 involves the equality of locations, and the encoding used here does not accommodate such tests. To better illustrate the LTS, examples are written in our language directly rather than using the syntactic sugar for references in Example 7. We use the standard Church numerals for encoding natural numbers and operators in the lambda calculus. In the bisimulation candidate relation, we explicitly describe these arithmetic encodings to avoid addressing issues of normal forms of Church numerals. In a larger application, the systematic way to proceed would be to move to a typed setting and enrich the transition system (and hence the bisimulation relation) with new constants.

Example 31 *(Local store).* This example shows that local declaration of a primitive pointcut and function at that primitive pointcut (providing local store) does not affect computation. Consider the following terms.

$$\mathsf{M} = \mathsf{x} \qquad \mathsf{N} = \mathbf{pcd}\ \mathsf{p}\,;\ \mathbf{fun}\ \mathsf{f@p} = \lambda.0\,;\ \mathsf{x}$$

We wish to prove that $\lambda x.M \sim \lambda x.N$. By congruence (Theorem 36) it suffices to show $M \sim N$. Define the relation $\mathscr{R}$ as

$$x;\cdot \vdash (\cdot/\cdot/x/\cdot) \;\; \mathscr{R} \;\; (\vec{A}/\cdot/x/\cdot)$$

where $\vec{A} = (\mathbf{pcd}\ p\,;\mathbf{fun}\ f@p = \lambda.0)$.

The only possible transition labels are $\mathsf{app}\ \phi$ and put.

$$\begin{array}{ccc} x;\cdot \vdash \cdot/\cdot/x/\cdot & \xrightarrow{\mathsf{app}\ \phi} & x,\phi;\cdot \vdash \cdot/\cdot/x\ \phi/\cdot \\ \mathscr{R}\ \big\{ & & \big\}\ \mathscr{R}^{\bullet}_{\mathtt{sf}} \\ x;\cdot \vdash \vec{A}/\cdot/x/\cdot & \xrightarrow{\mathsf{app}\ \phi} & x,\phi;\cdot \vdash \vec{A}/\cdot/x\ \phi/\cdot \end{array} \qquad \begin{array}{ccc} x;\cdot \vdash \cdot/\cdot/x/\cdot & \xrightarrow{\mathsf{put}} & x;\cdot \vdash \cdot/\cdot/x/x \\ \mathscr{R}\ \big\{ & & \big\}\ \mathscr{R}^{\bullet}_{\mathtt{sf}} \\ x;\cdot \vdash \vec{A}/\cdot/x/\cdot & \xrightarrow{\mathsf{put}} & x;\cdot \vdash \vec{A}/\cdot/x/x \end{array}$$

By Lemma 30, $x;\cdot \vdash (\cdot/\cdot/x/\cdot) \sim (\cdot/\cdot/\vec{A}\,;x/\cdot)$. □

Example 32 *(Higher-order functions).* This example demonstrates reasoning about a call to an unknown procedure. Define M and N as follows.

$$\begin{array}{l} M = x\,(\lambda.()\,);\,() \\ N = \mathbf{pcd}\ p\,;\mathbf{fun}\ f@p = \lambda.0\,; \\ \quad x\,(\lambda.(\mathbf{let}\ y = f\ ()\,;(\mathbf{adv}\ p = \lambda.\lambda.y+2)\,;()))\,; \\ \quad \mathbf{if}\ ((f\ ()\ \mathbf{mod}\ 2) = 0)\ \mathbf{then}\ ()\ \mathbf{else}\ \Omega \end{array}$$

In M, the external procedure x is invoked with a functional argument without side effects. In N, x is invoked with an argument that advises the local function f—corresponding to incrementing a local reference by two—thus, maintaining the invariant that a call to f yields an even number.

In our proof, we prove the local invariant of evenness separately, without referring to the external function call. The bisimulation principle allows us to modularly add the external function.

To prove $\lambda x.M \sim \lambda x.N$, it suffices to show that $M \sim N$ (as before, by congruence). We use the following definitions.

$$\begin{array}{rl} U &= \lambda.() \\ \mathscr{E} &= [-]\,;() \\ \vec{A} &= \mathbf{pcd}\ p\,;\mathbf{fun}\ f@p = \lambda.0 \\ V &= \lambda.(\mathbf{let}\ y = f\ ()\,;(\mathbf{adv}\ p = \lambda.\lambda.y+2)\,;()) \\ \mathscr{F} &= [-]\,;\mathbf{if}\ ((f\ ()\ \mathbf{mod}\ 2) = 0)\ \mathbf{then}\ ()\ \mathbf{else}\ \Omega \\ \vec{B}_0 &\text{ is the empty advice list} \\ \vec{B}_n &= \vec{B}_{n-1}\,;(\mathbf{adv}\ p = \lambda.\lambda.\mathsf{Dbl}_n) \\ \mathsf{Dbl}_0 &= \lambda f.\lambda x.x \\ \mathsf{Dbl}_{n+1} &= \lambda f.\lambda x.f(f(\mathsf{Dbl}_n\ f\ x)) \end{array}$$

Here, the term Dbl_n represents $2n$, although it is not syntactically identical to the Church numeral for $2n$, and we are making use of the convention mentioned earlier for application of a term (rather than application of a value). So, $M = \mathscr{E}[x\ U]$ and $N = \vec{A}\,;\mathscr{F}[x\ V]$. We first prove two purely local results without the external call to show that the tests under consideration (as given by $\mathscr{E},\mathscr{F}$) do not distinguish $(\vec{A},\vec{B}_m)$ and $(\vec{A},\vec{B}_n)$ for any m,n.

(1) $\cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V}$ and $\cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V}$ are bisimilar, for all m,n.
(2) $\cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V}$ and $\cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_n/\mathscr{E}/\mathsf{U}/\mathsf{U}$ are bisimilar, for all m,n.

We address (1); the proof for (2) is identical and omitted.

Let m,n range over all nonnegative integers. Define $\mathscr{R}$ as follows.

$$\cdot;\cdot \vdash (\vec{\mathsf{A}}, \vec{\mathsf{B}}_m/\mathscr{F}/\mathsf{V}/\mathsf{V}) \;\mathscr{R}\; (\vec{\mathsf{A}}, \vec{\mathsf{B}}_n/\mathscr{E}/\mathsf{V}/\mathsf{V})$$

There are three possibilities for the transition system labels. Let $\stackrel{\kappa}{\Longrightarrow}$ be the sequential composition of $\stackrel{\kappa}{\longrightarrow}$ and $\twoheadleftarrow\!\!\twoheadrightarrow$.

Case $\mathsf{put}, \mathsf{get}\,1$**:** For put, we have the following.

$$\begin{array}{ccc} \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V} & \xrightarrow{\mathsf{put}} & \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V},\mathsf{V} \\ \mathscr{R} \big\{ & & \big\} \mathscr{R}^\bullet_{\mathsf{dup}} \\ \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V} & \xrightarrow{\mathsf{put}} & \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V},\mathsf{V} \end{array}$$

Similarly for $\mathsf{get}\,1$.

Case $\mathsf{app}\,\phi$**:** Using the operational semantics,

$$\begin{array}{l} \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V} \xrightarrow{\mathsf{app}\,\phi} \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_{m+1}/\mathscr{E}/()/\mathsf{V} \\ \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V} \xrightarrow{\mathsf{app}\,\phi} \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_{n+1}/\mathscr{F}/()/\mathsf{V} \end{array}$$

and thus we have the following.

$$\begin{array}{ccc} \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V} & \stackrel{\mathsf{app}\,\phi}{\Longrightarrow} & \phi;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_{m+1}/\mathscr{E}/()/\mathsf{V} \\ \mathscr{R} \big\{ & & \big\} \mathscr{R}^\bullet \\ \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V} & \stackrel{\mathsf{app}\,\phi}{\Longrightarrow} & \phi;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_{n+1}/\mathscr{F}/()/\mathsf{V} \end{array}$$

Case $\mathsf{ret}\,\phi$**:** Use the invariant that for any m, the function call f () in advice context $\vec{\mathsf{A}}, \vec{\mathsf{B}}_m$ evaluates to an even number.

$$\begin{array}{ccc} \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V} & \stackrel{\mathsf{ret}\,\phi}{\Longrightarrow} & \phi;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_m/\cdot/()/\mathsf{V} \\ \mathscr{R} \big\{ & & \big\} \mathscr{R}^\bullet \\ \cdot;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V} & \stackrel{\mathsf{ret}\,\phi}{\Longrightarrow} & \phi;\cdot \vdash \vec{\mathsf{A}}, \vec{\mathsf{B}}_n/\cdot/()/\mathsf{V} \end{array}$$

This concludes the case analysis. Therefore, by Lemma 30, $\mathscr{R}^\bullet$, and hence $\mathscr{R}$ also, is contained in bisimilarity.

Now, using transitivity of bisimilarity on (1) and (2) yields

$$\cdot;\cdot \vdash (\vec{\mathsf{A}}/\mathscr{E}/\mathsf{U}/\cdot) \sim (\vec{\mathsf{A}}/\mathscr{F}/\mathsf{V}/\cdot).$$

Since x is not free in either configuration, we have

$$\mathsf{x};\cdot \vdash (\vec{\mathsf{A}}/\mathscr{E}/\mathsf{U}/\cdot) \sim (\vec{\mathsf{A}}/\mathscr{F}/\mathsf{V}/\cdot).$$

From Example 31, since we have that $\sim^{\bullet}_{\mathtt{sf}} \subseteq \sim$ and that $\mathscr{E}$ and U are state free for x:

$$\mathsf{x};\cdot \vdash (\vec{\mathsf{A}}/\mathscr{E}/\mathsf{U}/\cdot) \sim (\cdot/\mathscr{E}/\mathsf{U}/\cdot).$$

Using transitivity of $\sim$

$$\mathsf{x};\cdot \vdash (\vec{\mathsf{A}}/\mathscr{F}/\mathsf{V}/\cdot) \sim (\cdot/\mathscr{E}/\mathsf{U}/\cdot).$$

Since

$$(\mathsf{x};\cdot \vdash \cdot/\cdot/\mathscr{E}[\mathsf{x}\,\mathsf{U}]/\cdot) \xrightarrow{\mathsf{fcall}\,\mathsf{x}} (\mathsf{x};\cdot \vdash \cdot/\mathscr{E}/\mathsf{U}/\cdot), \text{ and}$$
$$(\mathsf{x};\cdot \vdash \cdot/\cdot/\vec{\mathsf{A}}\,;\mathscr{F}[\mathsf{x}\,\mathsf{V}]/\cdot) \xrightarrow{\mathsf{fcall}\,\mathsf{x}} (\mathsf{x};\cdot \vdash \vec{\mathsf{A}}/\mathscr{F}/\mathsf{V}/\cdot),$$

the required result,

$$\mathsf{x};\cdot \vdash (\cdot/\cdot/\vec{\mathsf{A}}\,;\mathscr{F}[\mathsf{x}\,\mathsf{V}]/\cdot) \sim (\cdot/\cdot/\mathscr{E}[\mathsf{U}]/\cdot),$$

follows since both sides have only weak $\mathsf{fcall}\,\mathsf{x}$ transitions to bisimilar targets. □

5 Results

Bisimilarity is sound and complete relative to observational congruence. Completeness is discussed in Appendix E. Here, we sketch a proof of soundness, focusing on the technical novelties of our analysis; details are deferred to the appendices.

In the rest of this section, we show that $\sim$ is a congruence. From this it is straightforward to establish soundness, i.e., that $M \sim N$ implies $M \equiv N$.

The key component of the proof is a substitution lemma that validates substitution of equals-for-equals for contexts that do not capture variables: the reader might want to view this semantically as an instance of the composition principles underlying game semantics [52, 53], and syntactically as our variant of the delayed substitutions of the SECD machine [69]. This is stated and proved in Appendix A.

5.1 Bisimulation Is a Congruence

The following notion of *compatibility* captures some useful properties of the initial configurations of Definition 20 and those reachable from them.

Definition 33. LTS configurations $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ are *compatible* if the following hold.

- All advice in Δ is symbolic advice of the form $\mathsf{adv}\ p = \alpha$.
- If $\mathsf{pcd}\ p \in \Delta$, then there exists $\mathsf{adv}\ p = \alpha \in \Delta$.
- For each $\mathsf{adv}\ p = \alpha \in \Delta$, it is the sole occurrence of α in Δ.
- If $\mathsf{fun}\ f\,@\,p = \phi \in \Delta$, then there exists $1 \leq i \leq \min(|\vec{U}|, |\vec{V}|)$ such that $\vec{U}_i = \vec{V}_i = f$ □

The next two lemmas provide the infrastructure required to reason separately about the active term and the remaining pieces of a configuration. Lemma 34 permits the substitution of identical terms for values in the active term spot of bisimilar configurations, while maintaining bisimilarity. Lemma 35 is dual.

Lemma 34 *(Inclusion of identical terms). Consider compatible configurations* $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U}$ *and* $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$ *and a term* L *such that* $fn(L) \subseteq \Gamma \cup dn(\Delta)$*. Then*

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$$

implies

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/L/\vec{V}.$$

Proof. See Appendix B. □

Lemma 35 *(Inclusion of identical contexts). Consider compatible configurations* $\Gamma;\Delta \vdash \cdot/\cdot/M/\vec{U}$ *and* $\Gamma;\Delta \vdash \cdot/\cdot/N/\vec{V}$ *and a well-formed LTS configuration* $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/()/\vec{W}$*, where no symbolic advice occurs in* M*,* $\vec{U}$*,* N*,* $\vec{V}$*. Then*

$$\Gamma;\Delta \vdash \cdot/\cdot/M/\vec{U} \sim \cdot/\cdot/N/\vec{V}$$

implies

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U},\vec{W} \sim \vec{A}/\vec{\mathscr{E}}/N/\vec{V},\vec{W}.$$

Proof. See Appendix C. □

Given this machinery, the proof that bisimulation is a congruence (and is therefore sound for contextual equivalence) is quite routine.

Theorem 36 *(Congruence of bisimilarity). Consider values* $U_1 \sim U_1'$*,* $U_2 \sim U_2'$ *and* $U \sim U'$ *and terms* $M \sim M'$*,* $M_1 \sim M_1'$ *and* $M_2 \sim M_2'$ *that contain no symbolic advice. Then we have the following.*

$$U_1\ U_2 \sim U_1'\ U_2' \qquad \lambda \mathsf{x}.M \sim \lambda \mathsf{x}.M'$$
$$\mathsf{let}\ \mathsf{x} = M_1\ ;\ M_2 \sim \mathsf{let}\ \mathsf{x} = M_1'\ ;\ M_2' \qquad \mathsf{fun}\ f@p = U\ ;\ M \sim \mathsf{fun}\ f@p = U'\ ;\ M'$$
$$\mathsf{pcd}\ p\ ;\ M \sim \mathsf{pcd}\ p\ ;\ M' \qquad \mathsf{adv}\ p = \lambda z.U\ ;\ M \sim \mathsf{adv}\ p = \lambda z.U'\ ;\ M'$$

Proof. See Appendix D. □

6 Conclusion

This paper is a step towards leveling the formal playing field between aspects and other programming paradigms. To our knowledge, we have presented the first description of bisimilarity for aspect languages. We contribute new operational techniques to show that bisimilarity is a congruence. Our results complement ongoing research in the aspect community on the design and implementation of aspect languages.

On one hand, our methods and techniques are those that are needed to address stateful higher-order programming languages. This is already seen in the basic format of our transition systems. Just as the interface of a (perhaps higher order) stateful program includes the global variables that are being used in the program, our LTS embodies the distinction between external (visible and advisable) pointcuts and the internal (hidden and unadvisable) pointcuts. Building further on analogies with higher-order

stateful computation, our bisimulation principle combines techniques used to address mobile processes (open bisimulation), names in the nu-calculus (via tracking leaked secrets in the LTS) and the lambda calculus (ENF-bisimulation). Our results suggest that from a purely theoretical viewpoint of studying general properties of the programming language, aspects are no more difficult to address formally than well-studied classical issues of higher-order imperative programs. We demonstrate the utility of our results by bridging the formal gap that exists between the foundations and the realizations of Open Modules. In particular, bisimulation is a congruence for a rich collection of temporal pointcuts including those that are realized in current implementations of Open Modules in aspect languages.

On the other hand, this message is tempered by its applicability to individual programs. Even theoretically speaking, our results do not directly yield a compositional translation of aspect programs into a higher order imperative language that preserves and reflects program equality. Our results are only a first step in terms of the practically important task of reasoning about concrete programs. We hope that they will serve as the conceptual infrastructure required to develop and validate "reasoning patterns" to address the common idioms of aspect-oriented programs.

Acknowledgements. We gratefully acknowledge comments by anonymous referees. We are particularly indebted to Kristian Støvring. His suggestions were key to clarifying the proof of soundness. Of course the authors are responsible for any errors and omissions in all proofs. James Riely was supported by NSF Career 0347542. Radha Jagadeesan and Corin Pitcher were supported by NSF Cybertrust 0430175.

Bibliography

[1] Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting object-interactions using composition-filters. In: Object-based distributed processing. LNCS (1993)

[2] Bergmans, L.: Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs. Ph.D. thesis, University of Twente (1994)

[3] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–355. Springer, Heidelberg (2001)

[4] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)

[5] Lieberherr, K.J.: Adaptive Object-Oriented Software: The Demeter method with propagation patterns. PWS Publishing Company (1996)

[6] Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the hyperspace approach. In: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development (2001)

[7] Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns (2000)

[8] Tarr, P.L., Ossher, H.: Hyper/J: Multi-dimensional separation of concerns for Java. In: ICSE, pp. 729–730 (2001)

[9] Coady, Y., Kiczales, G., Feeley, M.J., Smolyn, G.: Using AspectC to improve the modularity of path-specific customization in operating system code. In: ESEC/SIGSOFT FSE, pp. 88–98 (2001)
[10] Dutchyn, C., Tucker, D.B., Krishnamurthi, S.: Semantics and scoping of aspects in higher-order languages. Sci. Comput. Program. 63(3), 207–239 (2006)
[11] Walker, D., Zdancewic, S., Ligatti, J.: A theory of aspects. In: Runciman, C., Shivers, O. (eds.) ICFP, pp. 127–139. ACM, New York (2003)
[12] Sangiorgi, D.: Bisimulation: From the origins to today. In: LICS, pp. 298–302. IEEE Computer Society, Los Alamitos (2004)
[13] Sangiorgi, D.: The bisimulation proof method: Enhancements and open problems. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 18–19. Springer, Heidelberg (2006)
[14] Gordon, A.D.: Operational equivalences for untyped and polymorphic object calculi. In: Gordon, A.D., Pitts, A.M. (eds.) Higher-Order Operational Techniques in Semantics, Publications of the Newton Institute, pp. 9–54. Cambridge University Press, Cambridge (1998)
[15] Pitts, A.M.: Operationally-based theories of program equivalence. In: Dybjer, P., Pitts, A.M. (eds.) Semantics and Logics of Computation, Publications of the Newton Institute, pp. 241–298. Cambridge University Press, Cambridge (1997)
[16] Sumii, E., Pierce, B.C.: A bisimulation for type abstraction and recursion. J. ACM 54(5) (2007)
[17] Jeffrey, A., Rathke, J.: A theory of bisimulation for a fragment of concurrent ML with local names. Theor. Comput. Sci. 323(1-3), 1–48 (1999); preliminary version appeared in IEEE LICS 1999
[18] Koutavas, V., Wand, M.: Small bisimulations for reasoning about higher-order imperative programs. In: Morrisett and Jones [50], pp. 141–152
[19] Gordon, A.D., Rees, G.D.: Bisimilarity for a first-order calculus of objects with subtyping. In: POPL, pp. 386–395 (1996)
[20] Koutavas, V., Wand, M.: Bisimulations for untyped imperative objects. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 146–161. Springer, Heidelberg (2006)
[21] Meyer, A.R., Sieber, K.: Towards fully abstract semantics for local variables. In: POPL, pp. 191–203 (1988)
[22] Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: ICSE 2005: Proceedings of the 27th international conference on software engineering, pp. 49–58. ACM Press, New York (2005)
[23] Aldrich, J.: Open modules: Modular reasoning about advice. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 144–168. Springer, Heidelberg (2005)
[24] Ongkingco, N., Avgustinov, P., Tibble, J., Hendren, L., de Moor, O., Sittampalam, G.: Adding open modules to AspectJ. In: AOSD 2006: Proceedings of the 5th international conference on Aspect-oriented software development, pp. 39–50. ACM Press, New York (2006)
[25] Avgustinov, P., Bodden, E., Hajiyev, E., Hendren, L., Lhoták, O., de Moor, O., Ongkingco, N., Sereni, D., Sittampalam, G., Tibble, J.: Aspects for trace monitoring. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 20–39. Springer, Heidelberg (2006)
[26] Alur, R.: The benefits of exposing calls and returns. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 2–3. Springer, Heidelberg (2005)
[27] Alur, R., Madhusudan, P.: Adding nesting structure to words. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 1–13. Springer, Heidelberg (2006)
[28] Clifton, C., Leavens, G.T.: MiniMAO: An imperative core language for studying aspect-oriented reasoning. Sci. Comput. Program. 63(3), 321–374 (2006)

[29] Jagadeesan, R., Jeffrey, A., Riely, J.: An untyped calculus of aspect oriented programs. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743. Springer, Heidelberg (2003)
[30] Clifton, C., Leavens, G.T., Wand, M.: Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages (2003), http://www.cs.iastate.edu/~cclifton/papers/TR03-13.pdf
[31] Abadi, M., Cardelli, L.: A Theory of Objects. Springer, Heidelberg (1996)
[32] Rajan, H., Sullivan, K.J.: Classpects: unifying aspect- and object-oriented language design. In: Roman, G.-C., Griswold, W.G., Nuseibeh, B. (eds.) ICSE, pp. 59–68. ACM, New York (2005)
[33] Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic joinpoints in aspect-oriented programming. TOPLAS 26(5), 890–910 (2004)
[34] Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 46–60. Springer, Heidelberg (2003)
[35] Ligatti, J., Walker, D., Zdancewic, S.: A type-theoretic interpretation of pointcuts and advice. Sci. Comput. Program. 63(3), 240–266 (2006)
[36] Jagadeesan, R., Jeffrey, A., Riely, J.: Typed parametric polymorphism for aspects. Sci. Comput. Program. 63(3), 267–296 (2006)
[37] Li, H.C., Krishnamurthi, S., Fisler, K.: Modular verification of open features using three-valued model checking. Autom. Softw. Eng. 12(3), 349–382 (2005)
[38] Sihman, M., Katz, S.: Model checking applications of aspects and superimpositions. In: Foundations of Aspect Languages (2003)
[39] Ubayashi, N., Tamai, T.: Aspect-oriented programming with model checking. In: AOSD 2002: Proceedings of the 1st international conference on Aspect-oriented software development, pp. 148–154. ACM Press, New York (2002)
[40] Dantas, D.S., Walker, D.: Harmless advice. In: Morrisett and Jones [50], pp. 383–396
[41] Nakajima, S., Tamai, T.: Lightweight formal analysis of aspect-oriented models. In: UML 2004 Workshop on Aspect-Oriented Modeling (2004)
[42] Abramsky, S., Ong, C.-H.L.: Full abstraction in the lazy lambda calculus. Inf. Comput. 105(2), 159–267 (1993)
[43] Gordon, A.D.: Bisimilarity as a theory of functional programming. Electr. Notes Theor. Comput. Sci. 1 (1995)
[44] Howe, D.J.: Proving congruence of bisimulation in functional programming languages. Inf. Comput. 124(2), 103–112 (1996)
[45] Sumii, E., Pierce, B.C.: A bisimulation for dynamic sealing. Theor. Comput. Sci. 375(1-3), 169–192 (2007)
[46] Koutavas, V., Wand, M.: Proving class equivalence (submitted for publication) (July 2006)
[47] Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. In: LICS, pp. 293–302. IEEE Computer Society, Los Alamitos (2007)
[48] Sangiorgi, D.: A theory of bisimulation for the pi-calculus. Acta Inf. 33(1), 69–97 (1996)
[49] Lassen, S.: Eager normal form bisimulation. In: LICS, pp. 345–354. IEEE Computer Society Press, Los Alamitos (2005)
[50] Lassen, S.B.: Head normal form bisimulation for pairs and the $\lambda\mu$-calculus. In: LICS, pp. 297–306. IEEE Computer Society Press, Los Alamitos (2006)
[51] Støvring, K., Lassen, S.B.: A complete, co-inductive syntactic theory of sequential control and state. In: Hofmann, M., Felleisen, M. (eds.) POPL, pp. 161–172. ACM, New York (2007)
[52] Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. 163(2), 409–470 (2000)
[53] Hyland, J.M.E., Ong, C.-H.L.: On full abstraction for PCF: I, II, and III. Inf. Comput. 163(2), 285–408 (2000)

[54] Lassen, S.B., Levy, P.B.: Typed normal form bisimulation. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 283–297. Springer, Heidelberg (2007)
[55] Bockisch, C., Haupt, M., Mezini, M., Ostermann, K.: Virtual machine support for dynamic joinpoints. In: AOSD, pp. 83–92 (2004)
[56] Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
[57] Gordon, A.D.: A tutorial on co-induction and functional programming. In: Glasgow functional programming workshop, pp. 78–95. Springer, Heidelberg (1994)
[58] Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
[59] Felleisen, M., Friedman, D.P., Kohlbecker, E., Duba, B.: A syntactic theory of sequential control. Theor. Comput. Sci. 52(3), 205–237 (1987)
[60] De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34(1–2), 83–133 (1984)
[61] Jeffrey, A., Rathke, J.: Java Jr: Fully abstract trace semantics for a core Java language. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 423–438. Springer, Heidelberg (2005)
[62] Morris, J.H.: Lambda-Calculus Models of Programming Languages. PhD thesis, MIT (December 1968)
[63] Abadi, M., Fournet, C.: Access control based on execution history. In: Proceedings of the Network and Distributed System Security Symposium Conference (2003)
[64] Boebert, W.E., Kain, R.Y.: A practical alternative to hierarchical integrity policies. In: Proceedings of the Eighth National Computer Security Conference (1985)
[65] Walker, K.M., Sterne, D.F., Badger, M.L., Petkac, M.J., Shermann, D.L., Oostendorp, K.A.: Confining root programs with Domain and Type Enforcement (DTE). In: Proceedings of the Sixth USENIX UNIX Security Symposium (1996)
[66] Loscocco, P.A., Smalley, S.D.: Meeting critical security objectives with Security-Enhanced Linux. In: Proceedings of the 2001 Ottawa Linux Symposium (2001)
[67] Sangiorgi, D.: Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms. PhD thesis, University of Edinburgh (1993)
[68] Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River (1989)
[69] Landin, P.J.: The mechanical evaluation of expressions. Computer Journal 6(4), 308–320 (1964)
[70] Morrisett, J.G., Jones, S.L.P. (eds.): Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006. ACM, New York (2006)

A Substitution Lemma

This appendix develops the proof of Theorem 49, which states that substitution preserves bisimilarity. The substitutions that we consider provide two kinds of substitution information on a LTS configuration:

- Which value from the list of values in a configuration is substituted for a variable? This information is indicated by the positional index in $\vec{U}$ in a LTS configuration.
- Which contexts in the list of evaluation contexts are composed with other evaluation contexts and the active term? This information is specified by an integer stack.

Definition 37. An *extended substitution* σ consists of a partial function from variables (symbolic function names or symbolic advice names) to integers, a total order upon the domain of the partial function, and a nonempty stack of natural numbers.

Let ξ range over symbolic function names (metavariable ϕ) and symbolic advice names (metavariable α). If ξ is in the domain of the partial function of σ, we use $\sigma(\xi)$ for the value of the partial function of σ. We use $(\xi \mapsto k) \uplus \sigma$ for the operation of extending the domain of the partial function of σ to include ξ and placing ξ at the bottom of the total order on the domain of the new partial function. This operation is undefined if ξ is already in the domain of the original partial function σ.

In the nonempty stack of natural numbers, written $(\vec{m}, m)$ with m at the top, we require that $m_i > 0$, for all $1 \leq i \leq |\vec{m}|$, but allow $m = 0$. We write $|\sigma|$ for the length of the stack and $sum(\sigma)$ for the sum of the values on the stack. We sometimes regard a nonempty stack of natural numbers as the corresponding extended substitution with that stack and an empty partial function. □

Definition 38. Given contexts $\mathscr{E}$ and $\mathscr{F}$, define $\mathscr{E} \circ \mathscr{F}$ as $\mathscr{E}[\mathscr{F}]$. Observe that $\circ$ is associative, with $[-]$ as a left and right identity. Define:

$$Z(\mathscr{E}_1, \ldots, \mathscr{E}_n) \triangleq \mathscr{E}_1 \circ \cdots \circ \mathscr{E}_n$$

$$Z_{m_1,\ldots,m_k}(\mathscr{E}_{\langle 1,1\rangle}, \ldots, \mathscr{E}_{\langle 1,m_1\rangle}, \ \ldots, \ \mathscr{E}_{\langle k,1\rangle}, \ldots, \mathscr{E}_{\langle k,m_k\rangle}) \triangleq Z(\mathscr{E}_{\langle 1,1\rangle}, \ldots, \mathscr{E}_{\langle 1,m_1\rangle}), \ \ldots, \ Z(\mathscr{E}_{\langle k,1\rangle}, \ldots, \mathscr{E}_{\langle k,m_k\rangle})$$

Finally, define $Z_\sigma(\vec{\mathscr{E}}) \triangleq Z_{(\vec{m},m)}(\vec{\mathscr{E}})$ when σ's stack is $(\vec{m}, m)$. □

Note that $Z(\,\cdot\,) = [-]$ and that $Z_\sigma(\vec{\mathscr{E}})$ returns a sequence of contexts of length $|\sigma|$. Given that all but the rightmost element of σ must be nonzero, we may conclude that $|Z_\sigma(\vec{\mathscr{E}})| \leq |\vec{\mathscr{E}}| + 1$.

Definition 39. σ is *valid* for $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ if the following hold.

- The domain of σ is a subset of Γ.
- If α is an advice variable in the domain of σ, $U_{\sigma(\alpha)}$ is of the form $\lambda z.U$.
- If the total order for the domain of σ is $\xi_1, \ldots, \xi_n$, then, for all $1 \leq k \leq j \leq n$, we have ξ_k is not free in $U_{\sigma(\xi_j)}$.
- $sum(\sigma)$ equals the length of $\vec{\mathscr{E}}$. □

Definition 40. If σ is valid for $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$, we define:

$$[\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma \triangleq \Gamma';\Delta' \vdash \vec{A}'/\vec{\mathscr{E}}''/\mathscr{E}''[M']/\vec{U}''$$

where the primed elements on the right-hand side are derived as follows.

- Γ' is obtained from Γ by deleting every variable in the domain of σ.
- For every other metavariable χ, the single-primed version χ' is derived by substituting $U_{\sigma(\phi)}$ for ϕ in the configuration. The substitution is carried out following the total order of the variables given by σ (the least variable substituted first).
- $\vec{\mathscr{E}}'', \mathscr{E}'' = Z_\sigma(\mathscr{E}')$.
- $\vec{U}''$ is obtained from $\vec{U}'$ by deleting all values at positions in the substitution within σ.

In addition, we write $[\vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma$ when the context is uninteresting. □

For example, if $\mathbf{M} = \vec{A}/\mathscr{E}_1, \mathscr{E}_2, \mathscr{E}_3, \mathscr{E}_4, \mathscr{E}_5/M/\vec{U}$ then $[\mathbf{M}]_{3,2,0} = \vec{A}/\mathscr{E}_1[\mathscr{E}_2[\mathscr{E}_3]], \mathscr{E}_4[\mathscr{E}_5]/M/\vec{U}$, and $[\mathbf{M}]_{2,1,2} = \vec{A}/\mathscr{E}_1[\mathscr{E}_2], \mathscr{E}_3/\mathscr{E}_4[\mathscr{E}_5[M]]/\vec{U}$.

Definition 41. Write $\Gamma;\Delta \vdash \mathbf{M} \approx_\sigma \mathbf{N}$ if there exists $\Gamma';\Delta' \vdash \mathbf{M}' \sim \mathbf{N}'$ such that σ is valid for both $\Gamma';\Delta' \vdash \mathbf{M}'$ and $\Gamma';\Delta' \vdash \mathbf{N}'$, and we have $\Gamma;\Delta \vdash \mathbf{M} = [\Gamma';\Delta' \vdash \mathbf{M}']_\sigma$ and $\Gamma;\Delta \vdash \mathbf{N} = [\Gamma';\Delta' \vdash \mathbf{N}']_\sigma$. □

Two configurations are related by $\approx_\sigma$ if they are in the σ-image of configurations that are related by $\sim$. Theorem 49 formalizes the claim that bisimilarity is closed under substitution by stating that the relation $\approx = \bigcup_\sigma \approx_\sigma$ is a bisimulation.

To prove the substitution result, we first note that reduction is preserved by an extended substitution.

Lemma 42. *Given an extended substitution σ valid for $\Gamma;\Delta \vdash \mathbf{M}$, if:*

$$\Gamma;\Delta \vdash \mathbf{M} \xrightarrow{\tau} \Gamma;\Delta \vdash \mathbf{M}'$$

then:

$$[\Gamma;\Delta \vdash \mathbf{M}]_\sigma \xrightarrow{\tau} [\Gamma;\Delta \vdash \mathbf{M}']_\sigma$$

Proof. Reduction is preserved by substitution of values for variables in Γ, and by the addition of an evaluation context to a term (reduction is specified in terms of evaluation contexts).

Next we prove two lemmas that are used to choose bisimilar configurations in certain forms. Lemma 43 is used to eliminate trivial evaluation contexts (of the form $[-]$) from the evaluation context stack when they are applied directly to a value. Lemma 46 is used to remove substitutions that substitute one variable for another.

More specifically, Lemma 43 shows that for any $\approx$-related configurations, if the underlying pair of bisimilar configurations has a value on the left-hand side configuration, then there is another underlying pair of bisimilar configurations where the first nontrivial evaluation context from the evaluation context stack has been moved to the active term (to cause a reduction), or there is no nontrivial evaluation context. This manipulation is limited to the right-most m evaluation contexts when the stack from the extended substitution is $(\vec{m}, m)$. To indicate when a reduction is possible, we write $\Delta, \vec{A}/M \longrightarrow$ when there exist $\vec{B}$, N such that $\Delta, \vec{A}/M \longrightarrow \Delta, \vec{B}/N$.

Lemma 43. *Given an extended substitution σ valid for $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$, there exist Γ', $\vec{A}'$, $\vec{\mathscr{E}}'$, M', $\vec{U}'$, $\vec{B}'$, $\vec{\mathscr{F}}'$, N', $\vec{V}'$ and an extended substitution σ' valid for $\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \sim \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}'$ such that:*

1. $[\vec{A}/\vec{\mathscr{E}}/U/\vec{U}]_\sigma = [\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'}$
2. $[\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}]_\sigma \xrightarrow{\tau}{}^* [\Gamma';\Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'}$
3. *One of the following holds:*
   (a) ***Reduction:*** $\Delta, \vec{A}'/M' \longrightarrow$.
   (b) ***Value:*** *The σ' stack has the form $(\vec{m}, 0)$ and M' is a value.*

Proof. The proof is by induction on m, where the stack from σ is $(\vec{m}, m)$. If $m = 0$ the original extended substitution and configurations satisfy requirements (1)–(3), and we are done. If $m > 0$ then $\vec{\mathscr{E}}$ can be decomposed as $\vec{\mathscr{E}} = (\vec{\mathscr{E}}'', \mathscr{E}'')$, and the bisimilarity $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ yields the existence of $\vec{C}$ and V such that $\Delta, \vec{B}/N \twoheadrightarrow \Delta, \vec{C}/V$ and $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{C}/\vec{\mathscr{F}}/V/\vec{V}$. Now applying put then ret ϕ transitions to both sides (ϕ fresh), where $\vec{\mathscr{F}} = (\vec{\mathscr{F}}'', \mathscr{F}'')$, we have:

$$\Gamma, \phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}''/\mathscr{E}''[\phi]/\vec{U}, U \sim \vec{C}/\vec{\mathscr{F}}''/\mathscr{F}''[\phi]/\vec{V}, V$$

We obtain the extended substitution σ'' by replacing the stack of $(\phi \mapsto |\vec{U}|+1) \uplus \sigma$ with $(\vec{m}, m-1)$. Then $[\vec{A}/\vec{\mathscr{E}}''/\mathscr{E}''[\phi]/\vec{U}, U]_{\sigma''} = [\vec{A}/\vec{\mathscr{E}}/U/\vec{U}]_\sigma$ and $[\vec{C}/\vec{\mathscr{F}}''/\mathscr{F}''[\phi]/\vec{V}, V]_{\sigma''} = [\vec{C}/\vec{\mathscr{F}}/V/\vec{V}]_\sigma$. There are two cases depending on whether or not $\mathscr{E}'' = [-]$.

Case $\mathscr{E}'' \neq [-]$. When $\mathscr{E}'' \neq [-]$ there exists $\mathscr{E}'''$ such that $\mathscr{E}'' = \mathscr{E}'''[\mathsf{let\ x = [-]\ ;}\ M'']$, so we have the reduction:

$$\Delta, \vec{A}/\mathscr{E}''[\phi] = \Delta, \vec{A}/\mathscr{E}'''[\mathsf{let\ x = \phi\ ;}\ M''] \twoheadrightarrow \Delta, \vec{A}/\mathscr{E}'''[M''[\mathsf{x := \phi}]]$$

Thus, we define $\sigma' = \sigma''$, $\Gamma' = (\Gamma, \phi)$, and:

$$\begin{aligned}(\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}') &= (\vec{A}/\vec{\mathscr{E}}''/\mathscr{E}''[\phi]/\vec{U}, U)\\ (\vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}') &= (\vec{C}/\vec{\mathscr{F}}''/\mathscr{F}''[\phi]/\vec{V}, V)\end{aligned}$$

Requirements (1) and (3)(a) are satisfied immediately. Using Lemma 42 and the earlier reduction $\Delta, \vec{B}/N \twoheadrightarrow \Delta, \vec{C}/V$, we deduce that (2) is also satisfied:

$$\begin{aligned}& [\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}]_\sigma\\ \xrightarrow{\tau}{}^* \ & [\Gamma;\Delta \vdash \vec{C}/\vec{\mathscr{F}}/V/\vec{V}]_\sigma\\ = \ & [\Gamma, \phi;\Delta \vdash \vec{C}/\vec{\mathscr{F}}''/\mathscr{F}''[\phi]/\vec{V}, V]_{\sigma''}\\ = \ & [\Gamma';\Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'}\end{aligned}$$

Case $\mathscr{E}'' = [-]$. Now $\mathscr{E}''[\phi] = \phi$ is a value and the top element of the stack of σ'' is strictly smaller than that of σ, so the induction hypothesis can be applied to σ'' and $\Gamma, \phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}''/\phi/\vec{U}, U \sim \vec{C}/\vec{\mathscr{F}}''/\mathscr{F}''[\phi]/\vec{V}, V$ to yield σ' valid for $\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \sim \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}'$ such that:

- $[\vec{A}/\vec{\mathscr{E}}''/\phi/\vec{U}, U]_{\sigma''} = [\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'}$
- $[\Gamma, \phi;\Delta \vdash \vec{C}/\vec{\mathscr{F}}''/\mathscr{F}''[\phi]/\vec{V}, V]_{\sigma''} \xrightarrow{\tau}{}^* [\Gamma';\Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'}$

– • Either $\Delta, \vec{A}'/M' \longrightarrow$.
  • Or the σ' stack has the form $(\vec{m}, 0)$ and M' is a value.

For (1), by transitivity of equality, we have $[\vec{A}/\vec{\mathscr{E}}/U/\vec{U}]_\sigma = [\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'}$. For (2), using Lemma 42 and the earlier reduction $\Delta, \vec{B}/N \longrightarrow\!\!\!\!\rightarrow \Delta, \vec{C}/V$, we deduce that:

$$\begin{array}{rl} & [\Gamma; \Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}]_\sigma \\ \xrightarrow{\tau}{}^* & [\Gamma; \Delta \vdash \vec{C}/\vec{\mathscr{F}}/V/\vec{V}]_\sigma \\ = & [\Gamma, \phi; \Delta \vdash \vec{C}/\vec{\mathscr{F}}''/\mathscr{F}''[\phi]/\vec{V}, V]_{\sigma''} \\ \xrightarrow{\tau}{}^* & [\Gamma'; \Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'} \end{array}$$

Finally, (3) is satisfied immediately by the results of the induction hypothesis above. This completes the proof of Lemma 43.

To prove that substitutions of variables for variables can be eliminated, we introduce the notion of a variable chain length in a substitution and value list. This measures sequences of substitutions of a variable with another variable in the value list. Such sequences transfer control between values without making reductions. We formalize the notion of variable chain length as follows.

Definition 44. Given σ valid for $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$, and $\xi \in \Gamma$, define the natural number $\mathsf{varchain}(\xi, \sigma, \vec{U})$ by:

$$\mathsf{varchain}(\xi, \sigma, \vec{U}) \triangleq \begin{cases} 1 + \mathsf{varchain}(\psi, \sigma, \vec{U}) & \text{if } (\xi = \phi \wedge U_{\sigma(\phi)} = \psi) \\ & \quad \vee (\xi = \alpha \wedge U_{\sigma(\alpha)} = \lambda \mathsf{x}.\, \psi) \\ 1 + \mathsf{varchain}(\beta, \sigma, \vec{U}) & \text{if } (\xi = \phi \wedge U_{\sigma(\phi)} = \beta\texttt{<}U\texttt{>}) \\ & \quad \vee (\xi = \alpha \wedge U_{\sigma(\alpha)} = \lambda \mathsf{x}.\, \beta\texttt{<}U\texttt{>}) \\ 0 & \text{otherwise} \end{cases}$$

To simplify arguments about the two different kinds of call transitions, we introduce new terminology.

Definition 45. We say that $\Gamma; \Delta \vdash \mathbf{M}$ has a call transition on ξ if either $\Gamma; \Delta \vdash \mathbf{M} \xrightarrow{\mathsf{fcall}\ \xi}$ or $\Gamma; \Delta \vdash \mathbf{M} \xrightarrow{\mathsf{acall}\ \xi}$.

Lemma 46 shows that for any $\approx$-related configurations, if the underlying pair of bisimilar configurations has a call transition on the left-hand side configuration, then there is another underlying pair of bisimilar configurations where the call transition has turned into a reduction (due to substitutions from the value list for the variable in the call), or the call transition is unaffected by the substitution.

Lemma 46. *Consider an extended substitution σ valid for $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$. If $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ has a call transition on ξ then there exist Γ', $\vec{A}'$, $\vec{\mathscr{E}}'$, M', $\vec{U}'$, $\vec{B}'$, $\vec{\mathscr{F}}'$, N', $\vec{V}'$ and an extended substitution σ' valid for $\Gamma'; \Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \sim \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}'$ such that:*

1. $[\vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma = [\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'}$
2. $[\Gamma; \Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}]_\sigma \xrightarrow{\tau}{}^* [\Gamma'; \Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'}$

3. *One of the following holds:*
   (a) ***Reduction:*** $\Delta, \vec{A}'/M' \longrightarrow$.
   (b) ***Call:*** $\Gamma'; \Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$ *has a call transition on* ξ' *and* $\xi' \notin dom(\sigma')$.

Proof. If $\xi \notin dom(\sigma)$, the original extended substitution and configurations satisfy requirements (1)–(3), so there is nothing to prove. Otherwise $\xi \in dom(\sigma)$, and we proceed by induction on $\mathsf{varchain}(\xi, \sigma, \vec{U})$. The bisimilarity $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ allows us to deduce the existence of $\vec{C}$ and L such that $\Delta, \vec{B}/N \twoheadrightarrow \Delta, \vec{C}/L$, the configuration $\Gamma; \Delta \vdash \vec{C}/\vec{\mathscr{F}}/L/\vec{V}$ has a call transition on ξ, and $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{C}/\vec{\mathscr{F}}/L/\vec{V}$. There are two cases to consider: ξ may be an ordinary variable ϕ or an advice variable α.

Case ξ is an ordinary variable. If ξ is an ordinary variable ϕ, the call transition is $\mathsf{fcall}\ \phi$, and there exist evaluation contexts $\mathscr{E}$, $\mathscr{F}$ and values U, V such that $M = \mathscr{E}[\phi\ U]$, $L = \mathscr{F}[\phi\ V]$, and we have the following sequence (where ψ is fresh):

$$\begin{array}{rl} & \Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \\ = & \Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{E}[\phi\ U]/\vec{U} \\ \xrightarrow{\mathsf{fcall}\ \phi} & \Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U/\vec{U} \\ \xrightarrow{\mathsf{put}} & \Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U/\vec{U}, U \\ \xrightarrow{\mathsf{get}\ \sigma(\phi)} & \Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\phi)}/\vec{U}, U \\ \xrightarrow{\mathsf{app}\ \psi} & \Gamma, \psi; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\phi)}\ \psi/\vec{U}, U \end{array}$$

Since $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{C}/\vec{\mathscr{F}}/L/\vec{V}$ and $L = \mathscr{F}[\phi\ V]$, there is a corresponding sequence from $\Gamma; \Delta \vdash \vec{C}/\vec{\mathscr{F}}/L/\vec{V}$ that yields:

$$\Gamma, \psi; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\phi)}\ \psi/\vec{U}, U \sim \vec{C}/\vec{\mathscr{F}}, \mathscr{F}/V_{\sigma(\phi)}\ \psi/\vec{V}, V$$

We obtain the extended substitution σ'' by replacing the stack of $(\psi \mapsto |\vec{U}| + 1) \uplus \sigma$ with $(\vec{m}, m+1)$, where the stack from σ is $(\vec{m}, m)$. Then:

$$\begin{aligned} [\vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\phi)}\ \psi/\vec{U}, U]_{\sigma''} &= [\vec{A}/\vec{\mathscr{E}}/\mathscr{E}[U_{\sigma(\phi)}\ U]/\vec{U}]_{\sigma} \\ &= [\vec{A}/\vec{\mathscr{E}}/\mathscr{E}[\phi\ U]/\vec{U}]_{\sigma} \\ &= [\vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_{\sigma} \end{aligned}$$

Similarly, $[\vec{C}/\vec{\mathscr{F}}, \mathscr{F}/V_{\sigma(\phi)}\ \psi/\vec{V}, V]_{\sigma''} = [\vec{C}/\vec{\mathscr{F}}/L/\vec{V}]_{\sigma}$. There are two subcases, depending on whether or not the value $U_{\sigma(\phi)}$ is a λ-abstraction. In the first subcase, if $U_{\sigma(\phi)} = \lambda \mathsf{x}.M''$ is a λ-abstraction, then $\Delta, \vec{A}/(U_{\sigma(\phi)}\ \psi) \twoheadrightarrow \Delta, \vec{A}/(M''[\mathsf{x} := \psi])$. Therefore, we set $\sigma' = \sigma''$, $\Gamma' = (\Gamma, \psi)$, and:

$$\begin{aligned} (\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}') &= (\vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\phi)}\ \psi/\vec{U}, U) \\ (\vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}') &= (\vec{C}/\vec{\mathscr{F}}, \mathscr{F}/V_{\sigma(\phi)}\ \psi/\vec{V}, V) \end{aligned}$$

Requirements (1) and (3) are immediately satisfied. Requirement (2) follows using Lemma 42 and the earlier reduction $\Delta, \vec{B}/N \twoheadrightarrow \Delta, \vec{C}/L$:

$$\begin{array}{rl} & [\Gamma; \Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}]_{\sigma} \\ \xrightarrow{\tau}{}^{*} & [\Gamma; \Delta \vdash \vec{C}/\vec{\mathscr{F}}/L/\vec{V}]_{\sigma} \\ = & [\Gamma, \psi; \Delta \vdash \vec{C}/\vec{\mathscr{F}}, \mathscr{F}/V_{\sigma(\phi)}\ \psi/\vec{V}, V]_{\sigma''} \\ = & [\Gamma'; \Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'} \end{array}$$

This completes the first subcase. For the second subcase, since $U_{\sigma(\phi)}$ is not a λ-abstraction, it must be either a variable x or a symbolic advice call $\alpha\langle U''\rangle$. We define ξ'' by either $\xi'' = \mathsf{x}$ or $\xi'' = \alpha$ as appropriate. Then, $\Gamma, \phi; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\phi)}\ \psi/\vec{U}, U$ has a call transition on ξ''. Moreover, the variable chain length for ξ'' is strictly smaller than that for ϕ because:

$$\mathsf{varchain}(\phi, \sigma, \vec{U}) = 1 + \mathsf{varchain}(\xi'', \sigma, \vec{U}) = 1 + \mathsf{varchain}(\xi'', \sigma'', (\vec{U}, U))$$

Applying the induction hypothesis to σ'' and:

$$\Gamma, \psi; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\phi)}\ \psi/\vec{U}, U \sim \vec{C}/\vec{\mathscr{F}}, \mathscr{F}/V_{\sigma(\phi)}\ \psi/\vec{V}, V$$

yields σ' and: $\Gamma'; \Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \sim \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}'$ such that:

- $[\vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\phi)}\ \psi/\vec{U}, U]_{\sigma''} = [\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'}$
- $[\Gamma, \psi; \Delta \vdash \vec{C}/\vec{\mathscr{F}}, \mathscr{F}/V_{\sigma(\phi)}\ \psi/\vec{V}, V]_{\sigma''} \xrightarrow{\tau}{}^{*} [\Gamma'; \Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'}$
- - Either $\Delta, \vec{A}'/M' \longrightarrow$.
  - Or $\Gamma'; \Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$ has a call transition on ξ' and $\xi' \notin \mathit{dom}(\sigma')$.

Hence, requirement (3) is satisfied immediately, and requirement (1) follows by:

$$[\vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_{\sigma} = [\vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\phi)}\ \psi/\vec{U}, U]_{\sigma''} = [\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'}$$

Requirement (2) follows using Lemma 42 and the earlier reduction $\Delta, \vec{B}/N \twoheadrightarrow \Delta, \vec{C}/L$:

$$\begin{array}{rl} & [\Gamma; \Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}]_{\sigma} \\ \xrightarrow{\tau}{}^{*} & [\Gamma; \Delta \vdash \vec{C}/\vec{\mathscr{F}}/L/\vec{V}]_{\sigma} \\ = & [\Gamma, \psi; \Delta \vdash \vec{C}/\vec{\mathscr{F}}, \mathscr{F}/V_{\sigma(\phi)}\ \psi/\vec{V}, V]_{\sigma''} \\ \xrightarrow{\tau}{}^{*} & [\Gamma'; \Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'} \end{array}$$

This completes the second subcase (when $U_{\sigma(\phi)}$ is not a λ-abstraction), and also the case when ξ is a variable ϕ.

Case ξ is an advice variable. In the second case, ξ is an advice variable α, so the call transition is $\mathsf{acall}\ \alpha$, and there exist evaluation contexts $\mathscr{E}$, $\mathscr{F}$ and values U, V such that $M = \mathscr{E}[\alpha\langle W_1\rangle\ U]$ and $L = \mathscr{F}[\alpha\langle W_2\rangle\ V]$. In addition, σ is valid for both configurations, so both $U_{\sigma(\alpha)}$ and $V_{\sigma(\alpha)}$ must be abstractions with bodies that are also values, i.e., there exist W_3 and W_4 such that $U_{\sigma(\alpha)} = \lambda\mathsf{x}.W_3$ and $V_{\sigma(\alpha)} = \lambda\mathsf{x}.W_4$. Thus, we have the following sequence (where ψ_1, ψ_2 are fresh):

$$\begin{array}{rl} & \Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \\ = & \Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{E}[\alpha\langle W_1\rangle\ U]/\vec{U} \\ \xrightarrow{\mathsf{acall}\ \alpha} & \Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U/\vec{U}, W_1 \\ \xrightarrow{\mathsf{put}} & \Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U/\vec{U}, W_1, U \\ \xrightarrow{\mathsf{get}\ \sigma(\alpha)} & \Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\alpha)}/\vec{U}, W_1, U \\ \xrightarrow{\mathsf{app}\ \psi_1} & \Gamma, \psi_1; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/U_{\sigma(\alpha)}\ \psi_1/\vec{U}, W_1, U \\ = & \Gamma, \psi_1; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/(\lambda\mathsf{x}.W_3)\ \psi_1/\vec{U}, W_1, U \\ \xrightarrow{\tau} & \Gamma, \psi_1; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/W_3[\mathsf{x} := \psi_1]/\vec{U}, W_1, U \\ \xrightarrow{\mathsf{app}\ \psi_2} & \Gamma, \psi_1, \psi_2; \Delta \vdash \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/(W_3[\mathsf{x} := \psi_1])\ \psi_2/\vec{U}, W_1, U \end{array}$$

In the above we use the fact that $W_3[\mathsf{x} := \psi_1]$ is a value to justify the app ψ_2 transition. Since $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{C}/\vec{\mathscr{F}}/L/\vec{V}$ and $L = \mathscr{F}[\alpha{<}W_2{>}\, V]$, there is a corresponding sequence from $\Gamma;\Delta \vdash \vec{C}/\vec{\mathscr{F}}/L/\vec{V}$, and because reduction is deterministic we conclude:

$$\begin{aligned}\Gamma, \psi_1, \psi_2; \Delta \vdash\ & \vec{A}/\vec{\mathscr{E}}, \mathscr{E}/(W_3[\mathsf{x} := \psi_1])\ \psi_2/\vec{U}, W_1, U \\ \sim\ & \vec{C}/\vec{\mathscr{F}}, \mathscr{F}/(W_4[\mathsf{x} := \psi_1])\ \psi_2/\vec{V}, W_2, V\end{aligned}$$

We obtain the extended substitution σ'' by replacing the stack of the following extended substitution with $(\vec{m}, m+1)$, where the stack from σ is $(\vec{m}, m)$:

$$(\psi_2 \mapsto |\vec{U}| + 2) \uplus ((\psi_1 \mapsto |\vec{U}| + 1) \uplus \sigma)$$

The extended substitution σ'' is valid for the configurations above because ψ_1, ψ_2 were fresh. With this substitution we have:

$$\begin{aligned}& [\vec{A}/\vec{\mathscr{E}}, \mathscr{E}/(W_3[\mathsf{x} := \psi_1])\ \psi_2/\vec{U}, W_1, U]_{\sigma''} \\ &= [\vec{A}/\vec{\mathscr{E}}, \mathscr{E}/(W_3[\mathsf{x} := W_1])\ U/\vec{U}, W_1, U]_{\sigma''} \\ &= [\vec{A}/\vec{\mathscr{E}}/\mathscr{E}[(W_3[\mathsf{x} := W_1])\ U]/\vec{U}]_{\sigma} \\ &= [\vec{A}/\vec{\mathscr{E}}/\mathscr{E}[(W_3[\mathsf{x} := (W_1[\alpha := \lambda\mathsf{x}.W_3])])\ U]/\vec{U}]_{\sigma} \\ &= [\vec{A}/\vec{\mathscr{E}}/\mathscr{E}[\alpha{<}W_1{>}[\alpha := \lambda\mathsf{x}.W_3]\ U]/\vec{U}]_{\sigma} \\ &= [\vec{A}/\vec{\mathscr{E}}/\mathscr{E}[\alpha{<}W_1{>}[\alpha := U_{\sigma(\alpha)}]\ U]/\vec{U}]_{\sigma} \\ &= [\vec{A}/\vec{\mathscr{E}}/\mathscr{E}[\alpha{<}W_1{>}\, U]/\vec{U}]_{\sigma} \\ &= [\vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_{\sigma}\end{aligned}$$

Similarly:

$$[\vec{C}/\vec{\mathscr{F}}, \mathscr{F}/(W_4[\mathsf{x} := \psi_1])\ \psi_2/\vec{V}, W_2, V]_{\sigma''} = [\vec{C}/\vec{\mathscr{F}}/L/\vec{V}]_{\sigma}$$

There are two subcases, depending on whether or not the value $W_3[\mathsf{x} := \psi_1]$ is a λ-abstraction. Both subcases use the same reasoning as the corresponding subcases when ξ is an ordinary variable, i.e., when $W_3[\mathsf{x} := \psi_1]$ is a λ-abstraction, a beta reduction applies immediately and we are done; and when $W_3[\mathsf{x} := \psi_1]$ is not a λ-abstraction, the induction hypothesis is applied to σ'' and the bisimilar pair of configurations above, yielding the desired results by transitivity. This completes the case when ξ is an advice variable, and completes the proof of Lemma 46.

We now have an immediate corollary that is used to provide a suitable underlying bisimilar configuration when given a $\approx$-related configuration.

Corollary 47. *Given an extended substitution σ valid for $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$, there exist Γ', $\vec{A}'$, $\vec{\mathscr{E}}'$, M', $\vec{U}'$, $\vec{B}'$, $\vec{\mathscr{F}}'$, N', $\vec{V}'$ and an extended substitution σ' valid for $\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \sim \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}'$ such that:*

1. $[\vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_{\sigma} = [\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'}$
2. $[\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}]_{\sigma} \xrightarrow{\tau}{}^{*} [\Gamma';\Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'}$
3. *One of the following holds:*
   (a) ***Reduction:*** $\Delta, \vec{A}'/M' \longrightarrow$.
   (b) ***Value:*** *The σ' stack has the form $(\vec{m}, 0)$ and M' is a value.*
   (c) ***Call:*** *$\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$ has a call transition on ξ' and $\xi' \notin dom(\sigma')$.*

Proof. Consider $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$. If $\Delta,\vec{A}/M \longrightarrow$, the original extended substitution and configurations satisfy requirements (1)–(3), and we are done. Otherwise, M is a value or $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ has a call transition on some ξ. In the first case, the result follows immediately by Lemma 43. In the second case, the result follows immediately by Lemma 46. This completes the proof of Corollary 47.

Note that underlying reductions in (3)(a) of Corollary 47 can arise in three different ways. This corresponds to the different ways in which $\approx$-related configurations may have reductions.

Lemma 48 provides the core of the substitution result in Theorem 49. It shows that a weak transition from one side of a $\approx$-related configuration is matched by a weak transition from the other side, and the results are also $\approx$ related.

Lemma 48. *Given an extended substitution σ valid for $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$, if there is a weak $\kappa(\neq\tau)$ transition to a configuration* $\mathbf{M}$*:*

$$[\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma \xrightarrow{\kappa} \Gamma_1;\Delta_1 \vdash \mathbf{M}$$

then there exists a configuration $\mathbf{N}$ *such that:*

$$[\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}]_\sigma \xrightarrow{\kappa} \Gamma_1;\Delta_1 \vdash \mathbf{N}$$

and:

$$\Gamma_1;\Delta_1 \vdash \mathbf{M} \approx \mathbf{N}$$

Proof. By induction on the length of the reduction sequence in $[\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma \xrightarrow{\kappa} \Gamma_1;\Delta_1 \vdash \mathbf{M}$. Using Corollary 47, there exists an extended substitution σ' valid for $\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \sim \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}'$ such that:

- $[\vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma = [\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'}$
- $[\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}]_\sigma \xrightarrow{\tau}{}^* [\Gamma';\Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'}$
- One of the following holds:
  - ***Reduction:*** $\Delta,\vec{A}'/M' \longrightarrow$.
  - ***Value:*** The σ' stack has the form $(\vec{m},0)$ and M' is a value.
  - ***Call:*** $\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$ has a call transition on ξ' and $\xi' \notin dom(\sigma')$.

Then we have:

$$[\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'} = [\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma \xrightarrow{\kappa} \Gamma_1;\Delta_1 \vdash \mathbf{M}$$

We now consider the three possibilities given by Corollary 47: *Reduction*, *Value*, and *Call*. The *Reduction* case uses the induction hypothesis, and the *Value* and *Call* cases are the base cases where there are no reduction steps.

Case: Reduction. If $\Delta,\vec{A}'/M' \longrightarrow \Delta,\vec{A}''/M''$ for some $\vec{A}''$ and M'', then by Lemma 42 we have:

$$[\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'} \xrightarrow{\tau} [\Gamma';\Delta \vdash \vec{A}''/\vec{\mathscr{E}}'/M''/\vec{U}']_{\sigma'}$$

Moreover, reduction is deterministic, and the previous weak transition may be factored as:

$$[\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'} \xrightarrow{\tau} [\Gamma';\Delta \vdash \vec{A}''/\vec{\mathscr{E}}'/M''/\vec{U}']_{\sigma'} \xrightarrow{\kappa} \Gamma_1;\Delta_1 \vdash \mathbf{M}$$

Since reduction is contained in bisimilarity, we also deduce $\Gamma';\Delta \vdash \vec{A}''/\vec{\mathscr{E}}'/M''/\vec{U}' \sim \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}'$ from $\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \sim \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}'$. Thus, we may apply the induction hypothesis to the strictly shorter reduction sequence:

$$[\Gamma';\Delta \vdash \vec{A}''/\vec{\mathscr{E}}'/M''/\vec{U}']_{\sigma'} \xrightarrow{\kappa}\!\!\!\!\rightarrow \Gamma_1;\Delta_1 \vdash \mathbf{M}$$

This yields a configuration $\mathbf{N}$ such that:

$$[\Gamma';\Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'} \xrightarrow{\kappa}\!\!\!\!\rightarrow \Gamma_1;\Delta_1 \vdash \mathbf{N}$$

and:

$$\Gamma_1;\Delta_1 \vdash \mathbf{M} \approx \mathbf{N}$$

The result follows from the previous reduction sequence by transitivity:

$$[\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}]_{\sigma} \xrightarrow{\tau}{}^{*} [\Gamma';\Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'} \xrightarrow{\kappa}\!\!\!\!\rightarrow \Gamma_1;\Delta_1 \vdash \mathbf{N}$$

This completes the *Reduction* case.

Case: Value. If the σ' stack has the form $(\vec{m},0)$ and M' is a value, then there exist $\vec{C}'$ and V' such that $\Delta,\vec{B}'/N' \twoheadrightarrow \Delta,\vec{C}'/V'$ and:

$$\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \sim \vec{C}'/\vec{\mathscr{F}}'/V'/\vec{V}'$$

Next, define:

$$\Gamma'';\Delta'' \vdash \vec{A}''/\vec{\mathscr{E}}''/M''/\vec{U}'' \triangleq [\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'}$$
$$\Gamma'';\Delta'' \vdash \vec{B}''/\vec{\mathscr{F}}''/N''/\vec{V}'' \triangleq [\Gamma';\Delta \vdash \vec{C}'/\vec{\mathscr{F}}'/V'/\vec{V}']_{\sigma'}$$

Now the top of the stack of σ' is empty and M', V' are values, so M'' and N'' are also values. Therefore, κ must be a transition label with one of the following forms: $\mathsf{ret}\ \phi$, $\mathsf{app}\ \phi$, put, $\mathsf{get}\ i$, $\mathsf{fun}\, f @ p = \phi$, $\mathsf{adv}\ p = \alpha$ (where all names other p are fresh).

Subcase: $\kappa = \mathsf{ret}\ \phi$. In this subcase, we must have the decomposition $\vec{\mathscr{E}}'' = (\vec{\mathscr{E}}''', \mathscr{E}''')$, for some $\vec{\mathscr{E}}'''$, $\mathscr{E}'''$, so that:

$$\Gamma'';\Delta'' \vdash \vec{A}''/\vec{\mathscr{E}}''/M''/\vec{U}'' \xrightarrow{\mathsf{ret}\ \phi} \Gamma'',\phi;\Delta'' \vdash \vec{A}''/\vec{\mathscr{E}}'''/\mathscr{E}'''[\phi]/\vec{U}'' = \Gamma_1;\Delta_1 \vdash \mathbf{M}$$

Similarly, we must also have $\vec{\mathscr{F}}'' = (\vec{\mathscr{F}}''', \mathscr{F}''')$, for some $\vec{\mathscr{F}}'''$, $\mathscr{F}'''$, as well as the following transition:

$$\Gamma'';\Delta'' \vdash \vec{B}''/\vec{\mathscr{F}}''/N''/\vec{V}'' \xrightarrow{\mathsf{ret}\ \phi} \Gamma'',\phi;\Delta'' \vdash \vec{B}''/\vec{\mathscr{F}}'''/\mathscr{F}'''[\phi]/\vec{V}''$$

We then define $\mathbf{N} \triangleq (\vec{B}''/\vec{\mathscr{F}}'''/\mathscr{F}'''[\phi]/\vec{V}'')$. To establish $\Gamma_1;\Delta_1 \vdash \mathbf{M} \approx \mathbf{N}$, we also use $\mathsf{ret}\ \phi$ transitions on the underlying bisimilar configurations. Since $|\vec{\mathscr{E}}''| = |(\vec{\mathscr{E}}''', \mathscr{E}''')| > 0$, the stack of σ' must have the form $((\vec{m},m),0)$, where $m > 0$ by definition, so $|\vec{\mathscr{E}}'| = sum(\sigma') > 0$. Hence, $\vec{\mathscr{E}}' = (\vec{\mathscr{G}}_1, \mathscr{G}_1)$, for some $\vec{\mathscr{G}}_1$, $\mathscr{G}_1$, and we have the transition:

$$\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \xrightarrow{\mathsf{ret}\ \phi} \Gamma',\phi;\Delta \vdash \vec{A}'/\vec{\mathscr{G}}_1/\mathscr{G}_1[\phi]/\vec{U}'$$

By bisimilarity, we have $\vec{\mathscr{F}}' = (\vec{\mathscr{G}}_2, \mathscr{G}_2)$, for some $\vec{\mathscr{G}}_2, \mathscr{G}_2$, and the transition:

$$\Gamma'; \Delta \vdash \vec{C}'/\vec{\mathscr{F}}'/V'/\vec{V}' \xrightarrow{\mathsf{ret}\ \phi} \Gamma', \phi; \Delta \vdash \vec{C}'/\vec{\mathscr{G}}_2/\mathscr{G}_2[\phi]/\vec{V}'$$

such that:

$$\Gamma'; \Delta \vdash \vec{A}'/\vec{\mathscr{G}}_1/\mathscr{G}_1[\phi]/\vec{U}' \sim \vec{C}'/\vec{\mathscr{G}}_2/\mathscr{G}_2[\phi]/\vec{V}'$$

We obtain a new extended substitution σ'' from σ' by changing the stack from $((\vec{m}, m), 0)$ to $(\vec{m}, m-1)$ (recall that $m > 0$ so $m - 1 \geq 0$). The extended substitution σ'' is valid for the pair of bisimilar configurations above because $sum(\sigma') = sum(\sigma'') + 1$ and $|\vec{\mathscr{E}}'| = |\vec{\mathscr{G}}_1| + 1$, $|\vec{\mathscr{F}}'| = |\vec{\mathscr{G}}_2| + 1$. With the extended substitution σ'' we have:

$$\begin{array}{ll} & \mathbf{M} \\ = & (\vec{A}''/\vec{\mathscr{E}}'''/\mathscr{E}'''[\phi]/\vec{U}'') \\ = & [\vec{A}'/\vec{\mathscr{G}}_1/\mathscr{G}_1[\phi]/\vec{U}']_{\sigma''} \\ \approx & [\vec{C}'/\vec{\mathscr{G}}_2/\mathscr{G}_2[\phi]/\vec{V}']_{\sigma''} \\ = & (\vec{B}''/\vec{\mathscr{F}}'''/\mathscr{F}'''[\phi]/\vec{V}'') \\ = & \mathbf{N} \end{array}$$

Finally, by Lemma 42 and $\Delta, \vec{B}'/N' \twoheadrightarrow \Delta, \vec{C}'/V'$:

$$\begin{array}{ll} & [\Gamma'; \Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'} \\ \xrightarrow{\tau}{}^{*} & [\Gamma'; \Delta \vdash \vec{C}'/\vec{\mathscr{F}}'/V'/\vec{V}']_{\sigma'} \\ = & \Gamma''; \Delta \vdash \vec{B}''/\vec{\mathscr{F}}''/N''/\vec{V}'' \\ \xrightarrow{\mathsf{ret}\ \phi} & \Gamma_1; \Delta_1 \vdash \mathbf{N} \end{array}$$

Together with $\Gamma_1; \Delta_1 \vdash \mathbf{M} \approx \mathbf{N}$, this completes the subcase.

Subcases: $\kappa \in \{\mathsf{app}\ \phi, \mathsf{put}, \mathsf{get}\ i, \mathsf{fun}\ f @ p = \phi, \mathsf{adv}\ p = \alpha\}$. In each of these subcases, similar reasoning to that for the $\mathsf{ret}\ \phi$ subcase above applies. The (simplifying) difference is that these transitions leave the evaluation stack unchanged, and so $\sigma'' \triangleq \sigma'$.

We illustrate with the case for $\kappa = (\mathsf{adv}\ p = \alpha)$. We have for $p \in dn(\Delta'')$ and fresh α:

$$\Gamma''; \Delta'' \vdash \vec{A}''/\vec{\mathscr{E}}''/M''/\vec{U}'' \xrightarrow{\mathsf{adv}\ p=\alpha} \Gamma'', \alpha; \Delta'' \vdash \vec{A}''\,;\, \mathsf{adv}\ p = \alpha/\vec{\mathscr{E}}''/M''/\vec{U}'' = \Gamma_1; \Delta_1 \vdash \mathbf{M}$$

Similarly, we define $\mathbf{N} \triangleq (\vec{B}''\,;\, \mathsf{adv}\ p = \alpha/\vec{\mathscr{F}}''/N''/\vec{V}'')$, so:

$$\Gamma''; \Delta'' \vdash \vec{B}''/\vec{\mathscr{F}}''/N''/\vec{V}'' \xrightarrow{\mathsf{adv}\ p=\alpha} \Gamma'', \alpha; \Delta'' \vdash \vec{B}''\,;\, \mathsf{adv}\ p = \alpha/\vec{\mathscr{F}}''/N''/\vec{V}'' = \Gamma_1; \Delta_1 \vdash \mathbf{N}$$

Since M' and V' are values, and α is fresh, the same transition also applies to the underlying configurations:

$$\begin{array}{l} \Gamma'; \Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \xrightarrow{\mathsf{adv}\ p=\alpha} \Gamma', \alpha; \Delta \vdash \vec{A}'\,;\, \mathsf{adv}\ p = \alpha/\vec{\mathscr{E}}'/M'/\vec{U}' \\ \Gamma'; \Delta \vdash \vec{C}'/\vec{\mathscr{F}}'/V'/\vec{V}' \xrightarrow{\mathsf{adv}\ p=\alpha} \Gamma', \alpha; \Delta \vdash \vec{C}'\,;\, \mathsf{adv}\ p = \alpha/\vec{\mathscr{F}}'/V'/\vec{V}' \end{array}$$

yielding bisimilar configurations:

$$\Gamma', \alpha; \Delta \vdash \vec{A}'\,;\, \mathsf{adv}\ p = \alpha/\vec{\mathscr{E}}'/M'/\vec{U}' \sim \vec{C}'\,;\, \mathsf{adv}\ p = \alpha/\vec{\mathscr{F}}'/V'/\vec{V}'$$

The substitution σ' preserves the advice variable declaration and the symbolic advice:

$$\begin{aligned}\Gamma_1;\Delta_1 \vdash \mathbf{M} &= \Gamma'',\alpha;\Delta'' \vdash \vec{A}'' \,;\, \mathsf{adv}\ p=\alpha/\vec{\mathscr{E}}''/M''/\vec{U}'' \\ &= [\Gamma',\alpha;\Delta \vdash \vec{A}' \,;\, \mathsf{adv}\ p=\alpha/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'} \\ \Gamma_1;\Delta_1 \vdash \mathbf{N} &= \Gamma'',\alpha;\Delta'' \vdash \vec{B}'' \,;\, \mathsf{adv}\ p=\alpha/\vec{\mathscr{F}}''/N''/\vec{V}'' \\ &= [\Gamma',\alpha;\Delta \vdash \vec{C}' \,;\, \mathsf{adv}\ p=\alpha/\vec{\mathscr{F}}'/V'/\vec{V}']_{\sigma'}\end{aligned}$$

Hence, $\Gamma_1;\Delta_1 \vdash \mathbf{M} \approx \mathbf{N}$. Finally, by Lemma 42 and $\Delta,\vec{B}'/N' \twoheadrightarrow \Delta,\vec{C}'/V'$:

$$\begin{array}{rl} & [\Gamma';\Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'} \\ \xrightarrow{\tau}{}^{*} & [\Gamma';\Delta \vdash \vec{C}'/\vec{\mathscr{F}}'/V'/\vec{V}']_{\sigma'} \\ = & \Gamma'';\Delta \vdash \vec{B}''/\vec{\mathscr{F}}''/N''/\vec{V}'' \\ \xrightarrow{\mathsf{adv}\ p=\alpha} & \Gamma_1;\Delta_1 \vdash \mathbf{N}\end{array}$$

Together with $\Gamma_1;\Delta_1 \vdash \mathbf{M} \approx \mathbf{N}$, this completes the subcase for $\kappa = (\mathsf{adv}\ p=\alpha)$.

This completes the *Value* case.

Case: Call. $\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$ has a call transition on ξ' and $\xi' \notin dom(\sigma')$. The subcases when $\kappa = \mathsf{fcall}\ \phi$ and $\kappa = \mathsf{acall}\ \alpha$ are very similar. We present the latter subcase and omit the former.

Subcase: $\kappa = \mathsf{acall}\ \alpha$. Since there is a call transition on $\xi' = \alpha$, we have $M' = \mathscr{E}'[\alpha\langle W_1\rangle W_1']$ and:

$$\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \xrightarrow{\mathsf{acall}\ \alpha} \Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}',\mathscr{E}'/W_1'/\vec{U}',W_1$$

By bisimilarity we have $\Delta,\vec{B}'/N' \twoheadrightarrow \Delta,\vec{C}'/L'$, for $L' = \mathscr{F}'[\alpha\langle W_2\rangle W_2']$, and:

$$\Gamma';\Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}' \xrightarrow{\mathsf{acall}\ \alpha} \Gamma';\Delta \vdash \vec{C}'/\vec{\mathscr{F}}',\mathscr{F}'/W_2'/\vec{V}',W_2$$

such that:

$$\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}',\mathscr{E}'/W_1'/\vec{U}',W_1 \sim \vec{C}'/\vec{\mathscr{F}}',\mathscr{F}'/W_2'/\vec{V}',W_2$$

Now suppose that the stack of σ' is $(\vec{m},m)$ and that $Z_{\sigma'}(\vec{\mathscr{E}}') = (\vec{\mathscr{E}}'',\mathscr{E}'')$. We write $(\cdot)^{\dagger}$ for the result of applying the σ' substitutions from $\vec{U}'$ on a term, value list, evaluation context list, and declarations list. We obtain the new extended substitution σ'' from σ' by modifying the stack from $(\vec{m},m)$ to $((\vec{m},m+1),0)$, so $Z_{\sigma''}(\vec{\mathscr{E}}',\mathscr{E}') = (\vec{\mathscr{E}}'',\mathscr{E}''[\mathscr{E}'],[-])$. The extended substitution σ'' is valid for the bisimilar configurations above. Now, since $\alpha \notin dom(\sigma')$, we have for some Γ_1 (and with $\Delta_1 = \Delta^{\dagger}$):

$$\begin{array}{rl} & [\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_{\sigma'} \\ = & \Gamma_1;\Delta_1 \vdash \vec{A}'^{\dagger}/\vec{\mathscr{E}}''^{\dagger}/\mathscr{E}''[M']^{\dagger}/\vec{U}'^{\dagger} \\ = & \Gamma_1;\Delta_1 \vdash \vec{A}'^{\dagger}/\vec{\mathscr{E}}''^{\dagger}/\mathscr{E}''[\mathscr{E}'[\alpha\langle W_1\rangle W_1']]^{\dagger}/\vec{U}'^{\dagger} \\ \xrightarrow{\mathsf{acall}\ \alpha} & \Gamma_1;\Delta_1 \vdash \vec{A}'^{\dagger}/\vec{\mathscr{E}}''^{\dagger},(\mathscr{E}''[\mathscr{E}'])^{\dagger}/W_1'^{\dagger}/(\vec{U}',W_1)^{\dagger} \\ = & [\Gamma';\Delta \vdash \vec{A}'/\vec{\mathscr{E}}',\mathscr{E}'/W_1'/\vec{U}',W_1]_{\sigma''} \\ = & \Gamma_1;\Delta_1 \vdash \mathbf{M}\end{array}$$

Similarly, with the additional use of Lemma 42:

$$[\Gamma';\Delta \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}']_{\sigma'} \xrightarrow{\mathsf{acall}\ \alpha} [\Gamma';\Delta \vdash \vec{C}'/\vec{\mathscr{F}}',\mathscr{F}'/W_2'/\vec{V}',W_2]_{\sigma''}$$

Thus, we set $\mathbf{N} \triangleq [\Gamma'; \Delta \vdash \vec{C}'/\vec{\mathscr{F}}', \mathscr{F}'/W_2'/\vec{V}', W_2]_{\sigma''}$ and we know $\Gamma_1; \Delta_1 \vdash \mathbf{M} \approx \mathbf{N}$ from:

$$\Gamma'; \Delta \vdash \vec{A}'/\vec{\mathscr{E}}', \mathscr{E}'/W_1'/\vec{U}', W_1 \sim \vec{C}'/\vec{\mathscr{F}}', \mathscr{F}'/W_2'/\vec{V}', W_2$$

This completes the acall α subcase, the *Call* case, and the proof of Lemma 48.

Finally, we state and prove the substitution result.

Theorem 49 *(Substitution). The relation* $\approx \; = \bigcup_\sigma \approx_\sigma$ *is a bisimulation.*

Proof. We show that $\approx$ is a bisimulation by coinduction. Consider $\Gamma; \Delta \vdash \mathbf{M} \approx \mathbf{N}$. For the left-to-right direction, if there is a weak transition $\Gamma; \Delta \vdash \mathbf{M} \xRightarrow{\kappa'} \Gamma'; \Delta' \vdash \mathbf{M}'$ for $\kappa' \neq \tau$, then, by Lemma 48, there exists a configuration $\mathbf{N}'$ such that: $\Gamma; \Delta \vdash \mathbf{N} \xRightarrow{\kappa'} \Gamma'; \Delta' \vdash \mathbf{N}'$ and $\Gamma'; \Delta' \vdash \mathbf{M}' \approx \mathbf{N}'$. For the right-to-left direction, since bisimilarity is symmetric, $\approx$ is also symmetric. Hence, if there is a weak transition $\Gamma; \Delta \vdash \mathbf{N} \xRightarrow{\kappa''} \Gamma''; \Delta'' \vdash \mathbf{N}''$ for $\kappa'' \neq \tau$, then, by Lemma 48 and two uses of symmetry, there exists a configuration $\mathbf{M}''$ such that: $\Gamma; \Delta \vdash \mathbf{M} \xRightarrow{\kappa''} \Gamma''; \Delta'' \vdash \mathbf{M}''$ and $\Gamma''; \Delta'' \vdash \mathbf{M}'' \approx \mathbf{N}''$.

B Identity Extension for Terms

This section sketches the proof of Lemma 34. We first sketch an auxiliary lemma concerning the addition of a fresh public PCD and initial advice to bisimilar configurations.

Lemma 50. *If* $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ *and* p, α *are fresh, then:* $\Gamma, \alpha; \Delta, \mathsf{pcd}\ p, \mathsf{adv}\ p = \alpha \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$.

Proof. A straightforward bisimulation proof using Lemma 30. The bisimulation contains not only the configuration with the addition of α and $\mathsf{pcd}\ p, \mathsf{adv}\ p = \alpha$ but also functions and advice (at p) that can be added by the environment. The values resulting from looking up those functions are identical on both sides and state free, and thus Lemma 30 allows them to be safely ignored. □

For the proof sketch of Lemma 34 in the rest of this section, we assume that:

- $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$
- M, N are values.
- All names in $fn(L)$ are bound in $\Gamma; \Delta$.
- $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma; \Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ are compatible.

The proof proceeds by structural induction on L.

Case x, x *not bound in* Δ. Already established in Lemma 30, which, among other things, shows that bisimilarity is closed under replacement of the active term by a state free term with free variables bound in Γ. The variable x is such a state-free term.

Case f, f *bound in* Δ. $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma; \Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ have no τ-reductions.

The configurations are compatible, so let i be the index of the value list that has f in $\vec{U}$ and $\vec{V}$. Using transition get i, we get:

$$\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/f/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/f/\vec{V}$$

Case $U\ V$. The induction hypothesis on U yields:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/U/\vec{V}$$

and hence using put:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U},U \sim \vec{B}/\vec{\mathscr{F}}/U/\vec{V},U$$

Using induction hypothesis on V yields:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U},U \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V},U$$

and hence using put

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U},U,V \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V},U,V$$

The term $\mathsf{x}_1\ \mathsf{x}_2$ is state free, so Lemma 30 yields:

$$\Gamma,\mathsf{x}_1,\mathsf{x}_2;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathsf{x}_1\ \mathsf{x}_2/\vec{U},U,V \sim \vec{B}/\vec{\mathscr{F}}/\mathsf{x}_1\ \mathsf{x}_2/\vec{V},U,V$$

Consider substitution σ with stack $((1,\ldots,1),0)$ and partial function given by $\{\mathsf{x}_i \mapsto i+|\vec{U}| \mid i=1,2\}$. Using Theorem 49 yields the required result.

Case $\mathsf{pcd}\ p\,;L$. Applying Lemma 50 to:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$$

gives:

$$\Gamma,\alpha;\Delta,\mathsf{pcd}\ p,\mathsf{adv}\ p=\alpha \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$$

By the induction hypothesis on L:

$$\Gamma,\alpha;\Delta,\mathsf{pcd}\ p,\mathsf{adv}\ p=\alpha \vdash \vec{A}/\vec{\mathscr{E}}/L/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/L/\vec{V}$$

A bisimulation proof establishes:

$$\Gamma,\alpha;\Delta \vdash \vec{A},\mathsf{pcd}\ p,\mathsf{adv}\ p=\alpha/\vec{\mathscr{E}}/L/\vec{U} \sim \vec{B},\mathsf{pcd}\ p,\mathsf{adv}\ p=\alpha/\vec{\mathscr{F}}/L/\vec{V}$$

And, with $W=\lambda z.\lambda \mathsf{x}.z\ \mathsf{x}$, a second bisimulation proof using Lemma 30 yields:

$$\Gamma,\alpha;\Delta \vdash \vec{A},\mathsf{pcd}\ p,\mathsf{adv}\ p=\alpha/\vec{\mathscr{E}}/L/\vec{U},W \sim \vec{B},\mathsf{pcd}\ p,\mathsf{adv}\ p=\alpha/\vec{\mathscr{F}}/L/\vec{V},W$$

Substitution of W for α using Theorem 49 gives:

$$\Gamma;\Delta \vdash \vec{A},\mathsf{pcd}\ p,\mathsf{adv}\ p=W/\vec{\mathscr{E}}/L/\vec{U},W \sim \vec{B},\mathsf{pcd}\ p,\mathsf{adv}\ p=W/\vec{\mathscr{F}}/L/\vec{V},W$$

A final bisimulation proof shows:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathsf{pcd}\ p\,;L/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/\mathsf{pcd}\ p\,;L/\vec{V}$$

Case $\mathsf{fun}\, f@p = U\,;L$. Using induction on U we deduce that:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/U/\vec{V}$$

and hence:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U},U \sim \vec{B}/\vec{\mathscr{F}}/U/\vec{V},U$$

Using $\mathsf{fun}\, f@p = \phi$ on both sides, then the induction hypothesis on L, we get:

$$\Gamma,\phi;\Delta,\mathsf{fun}\, f@p = \phi \vdash \vec{A}/\vec{\mathscr{E}}/L/\vec{U},U \sim \vec{B}/\vec{\mathscr{F}}/L/\vec{V},U$$

Consider substitution σ with stack $((1,\ldots,1),0)$ and partial function given by $\{\phi \mapsto 1+|\vec{U}|\}$. Using Theorem 49, and a simple bisimulation proof to move the function declaration to the active term, yields the required result.

Case $\mathsf{adv}\ p = U\,;L$. Similar to above, but using $\mathsf{adv}\ p = \alpha$ transitions instead of $\mathsf{fun}\, f@p = \phi$.

Case $\mathsf{let}\, \mathsf{x} = L_1\,;L_2$. For a fixed L_2, define a bisimulation candidate $\mathscr{R}$ to be the least relation containing bisimilarity and such that:

1. If $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$, then $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathsf{let}\, \mathsf{x} = M\,;L_2/\vec{U}\ \mathscr{R}\ \vec{B}/\vec{\mathscr{F}}/\mathsf{let}\, \mathsf{x} = N\,;L_2/\vec{V}$.
2. If $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{E},\vec{\mathscr{E}}'/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}},\mathscr{F},\vec{\mathscr{F}}'/N/\vec{V}$, and $|\vec{\mathscr{E}}| = |\vec{\mathscr{F}}|$, $|\vec{\mathscr{E}}'| = |\vec{\mathscr{F}}'|$, then $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathsf{let}\, \mathsf{x} = \mathscr{E}\,;L_2,\vec{\mathscr{E}}'/M/\vec{U}\ \mathscr{R}\ \vec{B}/\vec{\mathscr{F}},\mathsf{let}\, \mathsf{x} = \mathscr{F}\,;L_2,\vec{\mathscr{F}}'/N/\vec{V}$.

We sketch the argument that $\mathscr{R}$ is in fact a bisimulation.

For (1), if $\Delta,\vec{A}/M \twoheadrightarrow \Delta,\vec{A}'/U$, then we can deduce $\Delta,\vec{B}/N \twoheadrightarrow \Delta,\vec{B}'/V$, for some $\vec{B}'$, V, and so we have both $\Delta,\vec{A}/\mathsf{let}\, \mathsf{x} = M\,;L_2 \twoheadrightarrow \Delta,\vec{A}'/L_2[\mathsf{x} := U]$ and $\Delta,\vec{B}/\mathsf{let}\, \mathsf{x} = N\,;L_2 \twoheadrightarrow \Delta,\vec{B}'/L_2[\mathsf{x} := V]$. We also have a bisimilarity $\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}'/\vec{\mathscr{F}}/V/\vec{V}$, and via introduction of x (assumed fresh, otherwise rename bound variables) and corresponding put transitions, we have $\Gamma,\mathsf{x};\Delta \vdash \vec{A}'/\vec{\mathscr{E}}/U/\vec{U},U \sim \vec{B}'/\vec{\mathscr{F}}/V/\vec{V},V$. Applying the induction hypothesis for L_2 yields $\Gamma,\mathsf{x};\Delta \vdash \vec{A}'/\vec{\mathscr{E}}/L_2/\vec{U},U \sim \vec{B}'/\vec{\mathscr{F}}/L_2/\vec{V},V$. Consider substitution σ with stack $((1,\ldots,1),0)$ and partial function given by $\{\mathsf{x} \mapsto 1+|\vec{U}|\}$. Using Theorem 49 yields $\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}/L_2[\mathsf{x} := U]/\vec{U} \sim \vec{B}'/\vec{\mathscr{F}}/L_2[\mathsf{x} := V]/\vec{V}$.

For (1), when a call transition with evaluation context $\mathscr{E}$ occurs during the reduction of $\Delta,\vec{A}/M$, there is a corresponding call transition (possibly after some reduction) with evaluation context $\mathscr{F}$ from $\Delta,\vec{B}/N$ due to the underlying bisimilarity. To reflect, in $\mathscr{R}$, the corresponding call transitions that take place for $\Delta,\vec{A}/\mathsf{let}\, \mathsf{x} = M\,;L_2$ and $\Delta,\vec{B}/\mathsf{let}\, \mathsf{x} = N\,;L_2$, (2) is used with evaluation contexts $\mathsf{let}\, \mathsf{x} = \mathscr{E}\,;L_2$ and $\mathsf{let}\, \mathsf{x} = \mathscr{F}\,;L_2$, respectively.

For (2), the interesting case is for $\mathsf{ret}\ \phi$ transitions when $\vec{\mathscr{E}}' = \vec{\mathscr{F}}' = ()$. In this case, the $\mathsf{let}\, \mathsf{x} = \mathscr{E}\,;L_2$ and $\mathsf{let}\, \mathsf{x} = \mathscr{F}\,;L_2$ contexts are restored and (1) is used to continue.

Finally, once $\mathscr{R}$ is known to be a bisimulation, we note that the induction hypothesis for L_1 yields:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L_1/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/L_1/\vec{V}$$

and so:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathsf{let}\, \mathsf{x} = L_1\,;L_2/\vec{U}\ \mathscr{R}\ \vec{B}/\vec{\mathscr{F}}/\mathsf{let}\, \mathsf{x} = L_1\,;L_2/\vec{V}$$

Since $\mathscr{R}$ is a bisimulation, we conclude:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathsf{let}\, \mathsf{x} = L_1\,;L_2/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/\mathsf{let}\, \mathsf{x} = L_1\,;L_2/\vec{V}$$

Case $\lambda x.L$. Consider the relation that consists of all compatible pairs of configurations $(\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M'/\vec{U},\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N'/\vec{V})$ such that there exists:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L_1/\vec{U}' \sim \vec{B}/\vec{\mathscr{F}}/L_2/\vec{V}'$$

such that:

- $\vec{U}'$ (resp. $\vec{V}'$) is obtained from $\vec{U}$ (resp. $\vec{V}$) by deleting all occurrences of $\lambda x.L$.
- The possibilities for L_1, L_2 are as follows
  - $L_1 = M'$ and $L_2 = N'$
  - Both M', N' are values, and one of the following holds:
    - $L_1 = L_2 = \lambda x.L$
    - $L_1 = L_2 = L[x := \phi]$

The required result follows from showing that this relation is a bisimulation. The straightforward proof to show this uses the induction hypothesis on L at all configurations having $L_1 = L_2 = L[x := \phi]$ in the active term position.

Inclusion of identical values and identical evaluation contexts. Since the first time when values from the value list (or contexts from the context list) can be moved into active position is when the term in the active position has become a value, and hence in the realm of applicability of Lemma 34, the addition of identical contexts and values can be performed in slightly more general situations.

Corollary 51 *(to Lemma 34). If:*

- $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$
- *All names in* $fn(U), \mathscr{E}, \mathscr{E}'$ *are bound in* Γ, Δ
- $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ *and* $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ *are compatible.*

then:

$$\Gamma;\Delta \vdash \vec{A}/\mathscr{E}', \vec{\mathscr{E}}, \mathscr{E}/M/\vec{U}, U \sim \vec{B}/\mathscr{E}', \vec{\mathscr{F}}, \mathscr{E}/N/\vec{V}, U$$

Corollary 51 is used in the proof of Lemma 35 given in Appendix C.

C Inclusion of Identical Contexts

In this section, we prove Lemma 35. The proof relies on the following auxiliary lemma that allows the addition of new common symbolic advice $\mathsf{adv}\ p = \beta$ before existing common symbolic advice $\mathsf{adv}\ p = \alpha$.

Lemma 52. *If:*

- $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$
- $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ *and* $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ *are compatible.*
- $\Delta = \Delta_1, \mathsf{adv}\ p = \alpha, \Delta_2$.
- α *does not occur in* Δ_1, Δ_2, $\vec{A}/\vec{\mathscr{E}}/M/\vec{U}$, $\vec{B}/\vec{\mathscr{F}}/N/\vec{V}$.

Then, for fresh β:

$$\Gamma, \beta; \Delta_1, \mathsf{adv}\ p = \beta, \mathsf{adv}\ p = \alpha, \Delta_2 \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$$

Proof. By bisimulation. We first define a function $\{|\}$ that replaces occurrences of α<U> with α<β<U>> in terms, values, contexts, and declarations, i.e., $\{|\}$ is the homomorphic, capture-avoiding extension of:

$$\{|\alpha\text{<}U\text{>}|\} = \alpha\text{<}\beta\text{<}\{|U|\}\text{>>}$$

Although $\{|\alpha|\} = \alpha$ this fact is not needed in the sequel because of the restrictions on occurrences of α. Consider $\vec{C}$ and L such that all occurrences of α in $\vec{C}$ and L have the form α<U> and where there are no occurrences of β. Then the reduction:

$$\Delta_1, \mathsf{adv}\ p = \alpha, \Delta_2, \vec{C}/L \longrightarrow \Delta_1, \mathsf{adv}\ p = \alpha, \Delta_2, \vec{C}'/L'$$

holds iff the following reduction holds:

$$\Delta_1, \mathsf{adv}\ p = \beta, \mathsf{adv}\ p = \alpha, \Delta_2, \{|\vec{C}|\}/\{|L|\} \longrightarrow \Delta_1, \mathsf{adv}\ p = \beta, \mathsf{adv}\ p = \alpha, \Delta_2, \{|\vec{C}'|\}/\{|L'|\}$$

The interesting case is for lookup of a function f defined at p, i.e., if the first lookup yields W, then the second lookup yields $\{|W|\}$:

$$(\Delta_1, \mathsf{adv}\ p = \beta, \mathsf{adv}\ p = \alpha, \Delta_2, \vec{C})(f) = \{|(\Delta_1, \mathsf{adv}\ p = \alpha, \Delta_2, \vec{C})(f)|\}$$

We also define a relation R between pairs of pairs of terms by:

$$\begin{array}{lll} R((M', N'), (M, N)) & \Leftrightarrow & (M' = M \wedge N' = N) \vee \\ & & (M \text{ and } N \text{ are values} \wedge M' = \beta\text{<}M\text{>} \wedge N' = \beta\text{<}N\text{>}) \vee \\ & & (M \text{ and } N \text{ are values} \wedge M' = \beta\text{<}M\text{>}\ \phi \wedge N' = \beta\text{<}N\text{>}\ \phi) \end{array}$$

Now define the bisimulation candidate $\mathscr{R}$ by:

$$\Gamma, \beta; \Delta_1, \mathsf{adv}\ p = \beta, \mathsf{adv}\ p = \alpha, \Delta_2 \vdash (\{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\}/M'/\vec{U}')\ \mathscr{R}\ (\{|\vec{B}|\}/\{|\vec{\mathscr{F}}|\}/N'/\vec{V}')$$

whenever:

1. $\Gamma; \Delta_1, \mathsf{adv}\ p = \alpha, \Delta_2 \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$
2. $\Gamma; \Delta_1, \mathsf{adv}\ p = \alpha, \Delta_2 \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma; \Delta_1, \mathsf{adv}\ p = \alpha, \Delta_2 \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ are compatible.
3. $R((M', N'), (\{|M|\}, \{|N|\}))$ and $R((U_i', V_i'), (\{|U_i|\}, \{|V_i|\}))$, for all $1 \le i \le |\vec{U}| = |\vec{V}| = |\vec{U}'| = |\vec{V}'|$.
4. β does not occur in Δ_1, Δ_2, $\vec{A}/\vec{\mathscr{E}}/M/\vec{U}$, $\vec{B}/\vec{\mathscr{F}}/N/\vec{V}$, and α only occurs in the form α<U>.

It can be verified that $\mathscr{R}$ is a bisimulation. We present the three nontrivial cases. For the first case, consider $M' = \mathscr{G}_1'[\alpha\text{<}W_1'\text{>}\ W_2']$, so the only transition possible is:

$$\begin{array}{l} \Gamma, \beta; \Delta_1, \mathsf{adv}\ p = \beta, \mathsf{adv}\ p = \alpha, \Delta_2 \vdash \{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\}/\mathscr{G}_1'[\alpha\text{<}W_1'\text{>}\ W_2']/\vec{U}' \xrightarrow{\mathsf{acall}\ \alpha} \\ \Gamma, \beta; \Delta_1, \mathsf{adv}\ p = \beta, \mathsf{adv}\ p = \alpha, \Delta_2 \vdash \{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\}, \mathscr{G}_1'/W_2'/\vec{U}', W_1' \end{array}$$

Since $\mathscr{G}_1'[\alpha\!<\!W_1'\!>\,W_2']$ is not a value, we know that $\mathscr{G}_1'[\alpha\!<\!W_1'\!>\,W_2'] = \{|M|\}$, so M must have form $\mathscr{G}_1[\alpha\!<\!W_1\!>\,W_2]$ and $\mathscr{G}_1' = \{|\mathscr{G}_1|\}$, $W_1' = \beta\!<\!\{|W_1|\}\!>$, and $W_2' = \{|W_2|\}$. The transition above is thus:

$$\Gamma,\beta;\Delta_1,\mathsf{adv}\ p=\beta,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\}/\{|\mathscr{G}_1|\}[\alpha\!<\!\beta\!<\!\{|W_1|\}\!>\,\{|W_2|\}\!>]/\vec{U}'$$
$$\xrightarrow{\mathsf{acall}\ \alpha}$$
$$\Gamma,\beta;\Delta_1,\mathsf{adv}\ p=\beta,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\},\{|\mathscr{G}_1|\}/\{|W_2|\}/\vec{U}',\beta\!<\!\{|W_1|\}\!>$$

In addition, we have the transition:

$$\Gamma;\Delta_1,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{G}_1[\alpha\!<\!W_1\!>\,W_2]/\vec{U} \xrightarrow{\mathsf{acall}\ \alpha}$$
$$\Gamma;\Delta_1,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{G}_1/W_2/\vec{U},W_1$$

By assumption:

$$\Gamma;\Delta_1,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{G}_1[\alpha\!<\!W_1\!>\,W_2]/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$$

Hence, there exist $\mathscr{G}_2$, W_3, and W_4 such that:

$$\Gamma;\Delta_1,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V} \xrightarrow{\mathsf{acall}\ \alpha}_{\twoheadrightarrow}$$
$$\Gamma;\Delta_1,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \vec{B}_1/\vec{\mathscr{F}},\mathscr{G}_2/W_4/\vec{V},W_3$$

And:

$$\Gamma;\Delta_1,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{G}_1/W_2/\vec{U},W_1 \sim \vec{B}_1/\vec{\mathscr{F}},\mathscr{G}_2/W_4/\vec{V},W_3$$

The preservation of reduction by $\{||\}$ yields:

$$\Gamma,\beta;\Delta_1,\mathsf{adv}\ p=\beta,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \{|\vec{B}|\}/\{|\vec{\mathscr{F}}|\}/\{|N|\}/\vec{V}' \xrightarrow{\mathsf{acall}\ \alpha}_{\twoheadrightarrow}$$
$$\Gamma,\beta;\Delta_1,\mathsf{adv}\ p=\beta,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \{|\vec{B}_1|\}/\{|\vec{\mathscr{F}}|\},\{|\mathscr{G}_2|\}/\{|W_4|\}/\vec{V}',\beta\!<\!\{|W_3|\}\!>$$

Finally, the fact that R allows introduction of β around top-level values establishes the case:

$$\Gamma,\beta;\Delta_1,\mathsf{adv}\ p=\beta,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash (\{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\},\{|\mathscr{G}_1|\}/\{|W_2|\}/\vec{U}',\beta\!<\!\{|W_1|\}\!>)$$
$$\mathscr{R}(\{|\vec{B}_1|\}/\{|\vec{\mathscr{F}}|\},\{|\mathscr{G}_2|\}/\{|W_4|\}/\vec{V}',\beta\!<\!\{|W_3|\}\!>)$$

For the second case, suppose M and N are values and $M' = \beta\!<\!\{|M|\}\!>$, $N' = \beta\!<\!\{|N|\}\!>$. The nontrivial transition is:

$$\Gamma,\beta;\Delta_1,\mathsf{adv}\ p=\beta,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\}/\beta\!<\!\{|M|\}\!>/\vec{U}' \xrightarrow{\mathsf{app}\ \phi}$$
$$\Gamma,\beta;\Delta_1,\mathsf{adv}\ p=\beta,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\}/\beta\!<\!\{|M|\}\!>\,\phi/\vec{U}'$$

There is a similar transition for $N' = \beta\!<\!\{|N|\}\!>$, and the targets are in the bisimulation $\mathscr{R}$ by definition of R. For the third case, suppose M and N are values and $M' = \beta\!<\!\{|M|\}\!>\,\phi$, $N' = \beta\!<\!\{|N|\}\!>\,\phi$. The only possible transition is:

$$\Gamma,\beta;\Delta_1,\mathsf{adv}\ p=\beta,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\}/\beta\!<\!\{|M|\}\!>\,\phi/\vec{U}' \xrightarrow{\mathsf{acall}\ \beta}$$
$$\Gamma,\beta;\Delta_1,\mathsf{adv}\ p=\beta,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\},[-]/\phi/\vec{U}',\{|M|\}$$

There is a similar transition for $N' = \beta\!<\!\{|N|\}\!>\,\phi$. By assumption:

$$\Gamma;\Delta_1,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$$

Using the definition of bisimulation in conjunction with Corollary 51, we can deduce:

$$\Gamma;\Delta_1,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash \vec{A}/\vec{\mathscr{E}},[-]/\phi/\vec{U},M \sim \vec{B}/\vec{\mathscr{F}},[-]/\phi/\vec{V},N$$

It follows that:

$$\begin{aligned}\Gamma;\Delta_1,\mathsf{adv}\ p=\beta,\mathsf{adv}\ p=\alpha,\Delta_2 \vdash\ & (\{|\vec{A}|\}/\{|\vec{\mathscr{E}}|\},[-]/\phi/\vec{U}',\{|M|\}) \\ & \mathscr{R}(\{|\vec{B}|\}/\{|\vec{\mathscr{F}}|\},[-]/\phi/\vec{V}',\{|N|\})\end{aligned}$$

Hence $\mathscr{R}$ is a bisimulation. The result follows immediately from the original hypothesis preventing occurrences of α, and that $\{||\}$ is the identity on such terms, values, contexts, and declarations. □

Proof of Lemma 35. For the proof of Lemma 35 in the rest of this section, a simple bisimulation proof allows us to assume without loss of generality that in $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/()/\vec{W}$, that is to be added to $\Gamma;\Delta \vdash \cdot/\cdot/M/\vec{U} \sim \cdot/\cdot/N/\vec{V}$, the declarations $\vec{A}$ have the form $\vec{A}_1,\vec{A}_2,\vec{A}_3$ where ($\vec{q}$ and $\vec{r}$ may be bound in $\vec{A}_1$ or Δ):

$$\begin{aligned}\vec{A}_1 &= \mathsf{pcd}\ \vec{p} \\ \vec{A}_2 &= \mathsf{fun}\ \vec{f}@\vec{q}=\vec{W}' \\ \vec{A}_3 &= \mathsf{adv}\ \vec{r}=\vec{W}''\end{aligned}$$

Repeated use of Lemma 50 and a simple bisimulation to move primitive pointcuts to the left of the common declaration list, allows $\vec{A}_1$ to be added along with initial symbolic advice $\vec{A}_4 = (\mathsf{adv}\ \vec{p}=\vec{\alpha})$ (where $\vec{\alpha}$ are fresh):

$$\Gamma,\vec{\alpha};\Delta,\vec{A}_1,\vec{A}_4 \vdash \cdot/\cdot/M/\vec{U} \sim \cdot/\cdot/N/\vec{V}$$

Note that this pair of configurations is also compatible. The next steps of the proof introduce symbolic function and advice bodies first, and then substitute their actual bodies from $\vec{A}_2$ and $\vec{A}_3$. First, function definitions with fresh symbolic bodies can be added using $\mathsf{fun}\ f@p=\phi$ transitions since the primitive pointcuts $\vec{p}$ are public, so with $\vec{A}_5 = (\mathsf{fun}\ \vec{f}@\vec{q}=\vec{\phi})$:

$$\Gamma,\vec{\alpha},\vec{\phi};\Delta,\vec{A}_1,\vec{A}_4,\vec{A}_5 \vdash \cdot/\cdot/M/\vec{U},\vec{f} \sim \cdot/\cdot/N/\vec{V},\vec{f}$$

Again the configurations are compatible. Next, we know there is no symbolic advice in M, $\vec{U}$, N, $\vec{V}$, $\vec{f}$, so for each $\mathsf{adv}\ r=W'' \in \vec{A}_3$ we successively apply Lemma 52 to the rightmost (necessarily symbolic) advice declaration for r in either Δ (if $\mathsf{pcd}\ r \in \Delta$) or $\vec{A}_4$ (if $\mathsf{pcd}\ r \in \vec{A}_1$). With variable renaming and declaration reordering, this yields fresh $\vec{\beta}$ and symbolic advice declarations $\vec{A}_6 = (\mathsf{adv}\ \vec{r}=\vec{\beta})$ such that:

$$\Gamma,\vec{\alpha},\vec{\phi},\vec{\beta};\Delta,\vec{A}_1,\vec{A}_4,\vec{A}_5,\vec{A}_6 \vdash \cdot/\cdot/M/\vec{U},\vec{f} \sim \cdot/\cdot/N/\vec{V},\vec{f}$$

Since the free names of $\vec{\mathscr{E}}$, $\vec{W}$, $\vec{W}'$, $\vec{W}''$ are bound in the context, we can use Corollary 51 to add values and evaluation contexts, yielding:

$$\Gamma,\vec{\alpha},\vec{\phi},\vec{\beta};\Delta,\vec{A}_1,\vec{A}_4,\vec{A}_5,\vec{A}_6 \vdash \cdot/\vec{\mathscr{E}}/M/\vec{U},\vec{f},\vec{W},\vec{W}',\vec{W}'' \sim \cdot/\vec{\mathscr{E}}/N/\vec{V},\vec{f},\vec{W},\vec{W}',\vec{W}''$$

The symbolic function declarations $\vec{A}_5$ and advice declarations $\vec{A}_6$ can be moved to the private declaration lists, simultaneously removing the $\vec{f}$ value lists, by a straightforward bisimulation proof to give the compatible bisimilar configurations:

$$\Gamma, \vec{\alpha}, \vec{\phi}, \vec{\beta}; \Delta, \vec{A}_1, \vec{A}_4 \vdash \vec{A}_5, \vec{A}_6/\vec{\mathscr{E}}/M/\vec{U}, \vec{W}, \vec{W}', \vec{W}'' \sim \vec{A}_5, \vec{A}_6/\vec{\mathscr{E}}/N/\vec{V}, \vec{W}, \vec{W}', \vec{W}''$$

Then the real function and advice bodies $\vec{W}'$ and $\vec{W}''$ can be substituted for $\vec{\phi}$ and $\vec{\beta}$, respectively, by Theorem 49 to recover $\vec{A}_2$ and $\vec{A}_3$:

$$\Gamma, \vec{\alpha}; \Delta, \vec{A}_1, \vec{A}_4 \vdash \vec{A}_2, \vec{A}_3/\vec{\mathscr{E}}/M/\vec{U}, \vec{W} \sim \vec{A}_2, \vec{A}_3/\vec{\mathscr{E}}/N/\vec{V}, \vec{W}$$

Now $\vec{A}_1, \vec{A}_4$ can be moved to the private declaration lists by a bisimulation proof:

$$\Gamma, \vec{\alpha}; \Delta \vdash \vec{A}_1, \vec{A}_4, \vec{A}_2, \vec{A}_3/\vec{\mathscr{E}}/M/\vec{U}, \vec{W} \sim \vec{A}_1, \vec{A}_4, \vec{A}_2, \vec{A}_3/\vec{\mathscr{E}}/N/\vec{V}, \vec{W}$$

Using Corollary 51 to add $|\vec{A}_4|$ copies of $\lambda z.z$ to both value lists, and Theorem 49 to substitute those advice bodies for $\vec{\alpha}$, we have:

$$\Gamma; \Delta \vdash \vec{A}_1, \mathsf{adv}\ \vec{p} = \lambda z.z, \vec{A}_2, \vec{A}_3/\vec{\mathscr{E}}/M/\vec{U}, \vec{W} \sim \vec{A}_1, \mathsf{adv}\ \vec{p} = \lambda z.z, \vec{A}_2, \vec{A}_3/\vec{\mathscr{E}}/N/\vec{V}, \vec{W}$$

However, it can be shown that the presence of advice declarations of the form $\mathsf{adv}\ p_i = \lambda z.z$ does not alter the result of advice lookup, so a bisimulation proof eliminates them to give:

$$\Gamma; \Delta \vdash \vec{A}_1, \vec{A}_2, \vec{A}_3/\vec{\mathscr{E}}/M/\vec{U}, \vec{W} \sim \vec{A}_1, \vec{A}_2, \vec{A}_3/\vec{\mathscr{E}}/N/\vec{V}, \vec{W}$$

This completes the proof of Lemma 35.

D Bisimulation Is a Congruence

This section contains the proof of Theorem 36 and shows that bisimulation is a congruence.

Application. Let $U_1 \sim U_1'$ and $U_2 \sim U_2'$. We need to show that $U_1\ U_2 \sim U_1'\ U_2'$. From:

$$\Gamma; \Delta \vdash \cdot/\cdot/U_2/\vec{g} \sim \cdot/\cdot/U_2'/\vec{g}$$

where $\vec{g}$ consists of the function names declared in Δ, and $fn(U_1\ U_2) \cup fn(U_1'\ U_2') \subseteq \Gamma \cup dn(\Delta)$, we deduce from put transitions that:

$$\Gamma; \Delta \vdash \cdot/\cdot/U_2/\vec{g}, U_2 \sim \cdot/\cdot/U_2'/\vec{g}, U_2'$$

From this, we deduce:

$$\Gamma, \mathsf{x}_1, \mathsf{x}_2; \Delta \vdash \cdot/\cdot/U_2/\vec{g}, U_2 \sim \cdot/\cdot/U_2'/\vec{g}, U_2'$$

and using Lemma 34:

$$\Gamma, \mathsf{x}_1, \mathsf{x}_2; \Delta \vdash \cdot/\cdot/\mathsf{x}_1\ \mathsf{x}_2/\vec{g}, U_2 \sim \cdot/\cdot/\mathsf{x}_1\ \mathsf{x}_2/\vec{g}, U_2'$$

Now, using Corollary 51 with a simple bisimulation proof to reorder value lists yields:

$$\Gamma, \mathsf{x}_1, \mathsf{x}_2; \Delta \vdash \cdot/\cdot/\mathsf{x}_1\ \mathsf{x}_2/\vec{g}, U_1', U_2 \sim \cdot/\cdot/\mathsf{x}_1\ \mathsf{x}_2/\vec{g}, U_1', U_2'$$

Similarly, from $\Gamma; \Delta \vdash \cdot/\cdot/U_1/ \sim \cdot/\cdot/U_1'/\cdot$ we get:

$$\Gamma, \mathsf{x}_1, \mathsf{x}_2; \Delta \vdash \cdot/\cdot/\mathsf{x}_1\ \mathsf{x}_2/\vec{g}, U_1, U_2 \sim \cdot/\cdot/\mathsf{x}_1\ \mathsf{x}_2/\vec{g}, U_1', U_2$$

Combining with transitivity:

$$\Gamma, \mathsf{x}_1, \mathsf{x}_2; \Delta \vdash \cdot/\cdot/\mathsf{x}_1\ \mathsf{x}_2/\vec{g}, U_1, U_2 \sim \cdot/\cdot/\mathsf{x}_1\ \mathsf{x}_2/\vec{g}, U_1', U_2'$$

Consider the substitution σ with stack $((1,\ldots,1),0)$ and partial function given by $\{\mathsf{x}_i \mapsto |\vec{g}| + i\}$. Theorem 49 yields:

$$\Gamma; \Delta \vdash [\cdot/\cdot/\mathsf{x}_1\ \mathsf{x}_2/\vec{g}, U_1, U_2]_\sigma \sim [\cdot/\cdot/\mathsf{x}_1\ \mathsf{x}_2/\vec{g}, U_1', U_2']_\sigma$$

and finishes the proof.

Function declaration. Let $U \sim U'$ and $M \sim M'$, where f occurs in neither U nor U'. We need to show that $\mathsf{fun}\ f@p = U\ ;\ M \sim \mathsf{fun}\ f@p = U'\ ;\ M'$.

We start with compatible LTS configurations:

$$\Gamma, f; \Delta \vdash \cdot/\cdot/U/\vec{g} \sim \cdot/\cdot/U'/\vec{g} \tag{1}$$

$$\Gamma, f; \Delta \vdash \cdot/\cdot/M/\vec{g} \sim \cdot/\cdot/M'/\vec{g} \tag{2}$$

Using put transitions with Eq. 1, we derive:

$$\Gamma, f; \Delta \vdash \cdot/\cdot/U/\vec{g}, U \sim \cdot/\cdot/U'/\vec{g}, U'$$

Apply Lemma 34 with term M to get:

$$\Gamma, f; \Delta \vdash \cdot/\cdot/M/\vec{g}, U \sim \cdot/\cdot/M/\vec{g}, U'$$

Next, adding the value U' to both sides of Eq. 2 using Corollary 51 yields:

$$\Gamma, f; \Delta \vdash \cdot/\cdot/M/\vec{g}, U' \sim \cdot/\cdot/M'/\vec{g}, U'$$

By transitivity, we have:

$$\Gamma, f; \Delta \vdash \cdot/\cdot/M/\vec{g}, U \sim \cdot/\cdot/M'/\vec{g}, U'$$

We can add a fresh variable ϕ:

$$\Gamma, f, \phi; \Delta \vdash \cdot/\cdot/M/\vec{g}, U \sim \cdot/\cdot/M'/\vec{g}, U'$$

Moreover, M, M', U, U' contain no symbolic advice by hypothesis, so Lemma 35 can be used to add the function declaration $\mathsf{fun}\ f'@p = \phi$ (where f' is fresh) and the value f' to both sides:

$$\Gamma, f, \phi; \Delta \vdash \mathsf{fun}\ f'@p = \phi/\cdot/M/\vec{g}, U, f' \sim \mathsf{fun}\ f'@p = \phi/\cdot/M'/\vec{g}, U', f'$$

Using Theorem 49 with substitution σ with stack $((1,\ldots,1),0)$ and partial function given by $\{\phi \mapsto |\vec{g}|+1, f \mapsto |\vec{g}|+2\}$ (using the fact that there are no occurrences of f in U, U') gives:

$$\Gamma;\Delta \vdash \mathsf{fun}\ f'@p = U/\cdot/(M[f := f'])/\vec{g} \sim \mathsf{fun}\ f'@p = U'/\cdot/(M'[f := f'])/\vec{g}$$

Since f' was fresh, there are no occurrences of f' in M or M'. Therefore, renaming the bound variable f' to f yields:

$$\Gamma;\Delta \vdash \mathsf{fun}\ f@p = U/\cdot/M/\vec{g} \sim \mathsf{fun}\ f@p = U'/\cdot/M'/\vec{g}$$

Now the function declarations can be moved into the term positions with a simple bisimulation:

$$\Gamma;\Delta \vdash \cdot/\cdot/\mathsf{fun}\ f@p = U\,;M/\vec{g} \sim \cdot/\cdot/\mathsf{fun}\ f@p = U'\,;M'/\vec{g}$$

Thus, we have:

$$\mathsf{fun}\ f@p = U\,;M \sim \mathsf{fun}\ f@p = U'\,;M'$$

Advice declaration. Let $U \sim U'$ and $M \sim M'$. We need to show that $\mathsf{adv}\ p = U\,;M \sim \mathsf{adv}\ p = U'\,;M'$.

We start with compatible LTS configurations:

$$\Gamma;\Delta \vdash \cdot/\cdot/U/\vec{g} \sim \cdot/\cdot/U'/\vec{g}$$

$$\Gamma;\Delta \vdash \cdot/\cdot/M/\vec{g} \sim \cdot/\cdot/M'/\vec{g}$$

where Δ contains the declaration $\mathsf{pcd}\ p$. As in the function declaration case, we derive:

$$\Gamma;\Delta \vdash \cdot/\cdot/M/\vec{g},U \sim \cdot/\cdot/M'/\vec{g},U'$$

Moreover, M, M', U, U' contain no symbolic advice by hypothesis, so a fresh α can be added to the context and Lemma 35 used to add the symbolic advice declaration $\mathsf{adv}\ p = \alpha$ to both sides:

$$\Gamma,\alpha;\Delta \vdash \mathsf{adv}\ p = \alpha/\cdot/M/\vec{g},U \sim \mathsf{adv}\ p = \alpha/\cdot/M'/\vec{g},U'$$

A simple bisimulation shows that the advice declarations can be moved into the terms:

$$\Gamma,\alpha;\Delta \vdash \cdot/\cdot/\mathsf{adv}\ p = \alpha\,;M/\vec{g},U \sim \cdot/\cdot/\mathsf{adv}\ p = \alpha\,;M'/\vec{g},U'$$

Finally, Theorem 49 with substitution σ with stack $((1,\ldots,1),0)$ and partial function given by $\{\alpha \mapsto |\vec{g}|+1\}$ gives:

$$\Gamma;\Delta \vdash \cdot/\cdot/\mathsf{adv}\ p = U\,;M/\vec{g} \sim \cdot/\cdot/\mathsf{adv}\ p = U'\,;M'/\vec{g}$$

Hence:

$$\mathsf{adv}\ p = U\,;M \sim \mathsf{adv}\ p = U'\,;M'$$

Lambda abstraction. Given $L \sim L'$, we wish to establish $\lambda x.L \sim \lambda x.L'$. The proof that the latter terms are bisimilar proceeds by a direct bisimulation argument.

Define a relation R between pairs of pairs of terms by:

$$R((M',N'),(M,N)) \quad \Leftrightarrow \quad (M' = M \wedge N' = N) \vee \\ (M \text{ and } N \text{ are values} \wedge M' = \lambda x.L \wedge N' = \lambda x.L') \vee$$

And define the bisimulation candidate $\mathscr{R}$ by:

$$\Gamma;\Delta \vdash (\vec{A}/\vec{\mathscr{E}}/M'/\vec{U}') \;\; \mathscr{R} \;\; (\vec{B}/\vec{\mathscr{F}}/N'/\vec{V}')$$

whenever:

1. $(fn(L) \cup fn(L')) \setminus \{x\} \subseteq \Gamma \cup dn(\Delta)$.
2. $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$
3. $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ are compatible.
4. $R((M',N'),(M,N))$ and $R((U_i',V_i'),(U_i,V_i))$, for all $1 \le i \le |\vec{U}| = |\vec{V}| = |\vec{U}'| = |\vec{V}'|$.

It can be verified that $\mathscr{R}$ is a bisimulation. The key case is for $\mathsf{app}\ \phi$ transitions from:

$$\Gamma;\Delta \vdash (\vec{A}/\vec{\mathscr{E}}/\lambda x.L/\vec{U}') \;\; \mathscr{R} \;\; (\vec{B}/\vec{\mathscr{F}}/\lambda x.L'/\vec{V}')$$

where we know that there exist values W, W' with compatible bisimilar configurations:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/W/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/W'/\vec{V}$$

such that $R((U_i',V_i'),(U_i,V_i))$, for all $1 \le i \le |\vec{U}| = |\vec{V}| = |\vec{U}'| = |\vec{V}'|$. We have to show that:

$$\Gamma;\Delta \vdash (\vec{A}/\vec{\mathscr{E}}/(\lambda x.L)\ \phi/\vec{U}') \;\; \mathscr{R} \;\; (\vec{B}/\vec{\mathscr{F}}/(\lambda x.L')\ \phi/\vec{V}')$$

Applying Lemma 34 to $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/W/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/W'/\vec{V}$ and $L[x := \phi]$, we get:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L[x := \phi]/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/L[x := \phi]/\vec{V} \tag{3}$$

In addition, by hypothesis and renaming variables, we have: $L[x := \phi] \sim L'[x := \phi]$. With Γ and Δ as above, the definition of $\sim$ implies, for function names $\vec{g}$ defined in Δ:

$$\Gamma;\Delta \vdash \cdot/\cdot/L[x := \phi]/\vec{g} \sim \cdot/\cdot/L'[x := \phi]/\vec{g}$$

Applying Lemma 35, to these compatible bisimilar configurations and the well-formed LTS configuration $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/()/\vec{V}$, together with a simple bisimulation to remove duplicated copies of $\vec{g}$, we have:

$$\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/L[x := \phi]/\vec{V} \sim \vec{B}/\vec{\mathscr{F}}/L'[x := \phi]/\vec{V} \tag{4}$$

Combining Eqs. 3 and 4 by transitivity yields:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L[x := \phi]/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/L'[x := \phi]/\vec{V}$$

Hence:

$$\Gamma;\Delta \vdash (\vec{A}/\vec{\mathscr{E}}/L[x := \phi]/\vec{U}') \;\; \mathscr{R} \;\; (\vec{B}/\vec{\mathscr{F}}/L'[x := \phi]/\vec{V}')$$

Finally, we have the reductions $\Delta,\vec{A}/(\lambda x.L)\ \phi \longrightarrow \Delta,\vec{A}/L[x := \phi]$ and $\Delta,\vec{A}/(\lambda x.L')\ \phi \longrightarrow \Delta,\vec{A}/L'[x := \phi]$, and reduction is deterministic, so:

$$\Gamma;\Delta \vdash (\vec{A}/\vec{\mathscr{E}}/(\lambda x.L)\ \phi/\vec{U}') \;\; \mathscr{R} \;\; (\vec{B}/\vec{\mathscr{F}}/(\lambda x.L')\ \phi/\vec{V}')$$

Thus, $\mathscr{R}$ is a bisimulation. It follows immediately that $\lambda x.L \sim \lambda x.L'$.

Let. Given $L_1 \sim L_1'$ and $L_2 \sim L_2'$, we must show that $\mathsf{let}\ \mathsf{x} = L_1\ ;\ L_2 \sim \mathsf{let}\ \mathsf{x} = L_1'\ ;\ L_2'$. We do this in two stages and then use the transitivity of $\sim$:

$$\mathsf{let}\ \mathsf{x} = L_1\ ;\ L_2 \sim \mathsf{let}\ \mathsf{x} = L_1'\ ;\ L_2$$
$$\mathsf{let}\ \mathsf{x} = L_1'\ ;\ L_2 \sim \mathsf{let}\ \mathsf{x} = L_1'\ ;\ L_2'$$

For the first stage, we use the bisimulation constructed in the Let case of the proof of Lemma 34 to deduce $\mathsf{let}\ \mathsf{x} = L_1\ ;\ L_2 \sim \mathsf{let}\ \mathsf{x} = L_1'\ ;\ L_2$ from $L_1 \sim L_1'$.

For the second stage, a bisimulation proof establishes:

$$\mathsf{let}\ \mathsf{x} = L_1'\ ;\ L_2 \sim \mathsf{let}\ \mathsf{x} = L_1'\ ;\ (\lambda \mathsf{x}.L_2)\ \mathsf{x}$$
$$\mathsf{let}\ \mathsf{x} = L_1'\ ;\ L_2' \sim \mathsf{let}\ \mathsf{x} = L_1'\ ;\ (\lambda \mathsf{x}.L_2')\ \mathsf{x}$$

Now we know from the previous case that $L_2 \sim L_2'$ implies $\lambda \mathsf{x}.L_2 \sim \lambda \mathsf{x}.L_2'$. After saving those values into the value lists, use Lemma 34 to add $\mathsf{let}\ \mathsf{x} = L_1'\ ;\ \mathsf{y}\ \mathsf{x}$ in the term position in both configurations. Substitution (Theorem 49) establishes the relationship between configurations with $(\mathsf{let}\ \mathsf{x} = L_1'\ ;\ \mathsf{y}\ \mathsf{x})[\mathsf{y} := \lambda \mathsf{x}.L_2]$ and $(\mathsf{let}\ \mathsf{x} = L_1'\ ;\ \mathsf{y}\ \mathsf{x})[\mathsf{y} := \lambda \mathsf{x}.L_2']$ in the term positions.

Primitive pointcuts declaration. Given $L \sim L'$, we must show that $\mathsf{pcd}\ p\,;\ L \sim \mathsf{pcd}\ p\,;\ L'$. From $L \sim L'$ we know:

$$\Gamma, \alpha; \Delta, \mathsf{pcd}\ p, \mathsf{adv}\ p = \alpha \vdash \cdot/\cdot/L/\cdot \sim \cdot/\cdot/L'/\cdot$$

A straightforward bisimulation shows that the declarations $\mathsf{pcd}\ p$ and $\mathsf{adv}\ p = \alpha$ can be moved to the front of the terms L and L'. Then the advice $\mathsf{adv}\ p = \alpha$ can be eliminated by substitution of $\lambda z.\lambda \mathsf{x}.z\,\mathsf{x}$ for α and a bisimulation proof, to leave:

$$\Gamma; \Delta \vdash \cdot/\cdot/\mathsf{pcd}\ p\,;\ L/\cdot \sim \cdot/\cdot/\mathsf{pcd}\ p\,;\ L'/\cdot$$

E Completeness

We show completeness ($M \equiv N$ implies $M \sim N$) by demonstrating the contrapositive ($M \not\sim N$ implies $M \not\equiv N$). The proof is a definability argument: we show that for every distinguishing trace, we can construct a context that witnesses the trace. This construction proceeds via an analysis of normal forms for such traces.

We first define *normal* traces and demonstrate that they are sufficient to distinguish nonbisimilar terms. (Because the language is deterministic, bisimilarity coincides with trace equivalence, which simplifies the argument.)

Let s, t range over *traces* of visible labels $s, t ::= \kappa_1, \ldots, \kappa_n$, with empty trace ε.

Definition 53. A *complete normal trace* is a trace that is generable by the following grammar over labels:

$$\begin{aligned}
\textsc{start} &::= \textsc{term}^*, \mathsf{put}, \textsc{ctxt}^* \\
\textsc{term} &::= \mathsf{fcall}\ \phi, \mathsf{put}, \textsc{ctxt}^*, \mathsf{ret}\ \psi \mid \mathsf{acall}\ \alpha, \mathsf{put}, \textsc{ctxt}^*, \mathsf{ret}\ \psi \\
\textsc{ctxt} &::= \mathsf{get}\ i, \mathsf{app}\ \phi, \textsc{term}^*, \mathsf{put} \mid \mathsf{fun}\ f\,@\,p = \phi \mid \mathsf{adv}\ p = \alpha
\end{aligned}$$

A *normal trace* is a prefix of a complete normal trace. □

Proposition 54. *If* $\Gamma;\Delta \vdash \mathbf{M} \lesssim \mathbf{N}$ *and* $\Gamma;\Delta \vdash \mathbf{M} \xrightarrow{s}\!\!\!\!\rightarrow$ *then* $\Gamma;\Delta \vdash \mathbf{N} \xrightarrow{s}\!\!\!\!\rightarrow$.

Proof By induction on the length of the trace. □

Proposition 55. *If* $\Gamma;\Delta \vdash \mathbf{M} \not\lesssim \mathbf{N}$ *then for some normal trace* s *and label* $\kappa \in \{\mathsf{fcall}, \mathsf{acall}\}$*:* $\Gamma;\Delta \vdash \mathbf{M} \xrightarrow{s}\!\!\!\!\rightarrow \xrightarrow{\kappa}\!\!\!\!\rightarrow$ *and* $\Gamma;\Delta \vdash \mathbf{N} \xrightarrow{s}\!\!\!\!\rightarrow \not\xrightarrow{\kappa}\!\!\!\!\rightarrow$.

Proof By an analysis of the commutativity of various labels and the resulting configurations. The essential observation is that the following categories of LTS states are disjoint:

- Write $\Gamma;\Delta \vdash \mathbf{M} \Uparrow$ if $\Gamma;\Delta \vdash \mathbf{M} \longrightarrow^{\omega}$.
- Write $\Gamma;\Delta \vdash \mathbf{M} \Downarrow \textsc{term}$ if $\Gamma;\Delta \vdash \mathbf{M} \xrightarrow{\kappa}\!\!\!\!\rightarrow$ for $\kappa \in \{\mathsf{fcall}, \mathsf{acall}\}$.
- Write $\Gamma;\Delta \vdash \mathbf{M} \Downarrow \textsc{ctxt}$ if $\Gamma;\Delta \vdash \mathbf{M} \xrightarrow{\kappa}\!\!\!\!\rightarrow$ for $\kappa \in \{\mathsf{put}, \mathsf{get}, \mathsf{ret}, \mathsf{app}, \mathsf{fun}, \mathsf{adv}\}$. □

Completeness indicates that bisimilarity is not too fine a relation. If two terms are not bisimilar, completeness requires that there be some context that distinguishes them. Following Definition 5, the context must *signal* in one case, but not the other (by calling the distinguished function signal). Recall that we write $M\!\downarrow$ if $M \longrightarrow\!\!\!\!\rightarrow \mathscr{E}[\mathsf{signal}\ U]$ for some evaluation context $\mathscr{E}$ and value U.

Given a distinguishing trace t, we show how to create a context $\mathbb{C}_t[\text{–}]$, such that $\mathbb{C}_t[\Gamma;\Delta \vdash \mathbf{M}]$ will signal exactly when $\Gamma;\Delta \vdash \mathbf{M}$ can perform trace t. The inductive argument requires that we define contexts of the form $\mathbb{C}^s_t[\text{–}]$, where actions s are completed and t have yet to be performed. We define the contexts to have the following properties.

Proposition 56. *Let* s, κ, t *be a normal trace. If* $\Gamma;\Delta \vdash \mathbf{M} \xrightarrow{\kappa}\!\!\!\!\rightarrow \Gamma';\Delta' \vdash \mathbf{M}'$ *then:*

$$\mathbb{C}^s_{\kappa,t}[\Gamma;\Delta \vdash \mathbf{M}] \longrightarrow\!\!\!\!\rightarrow \mathbb{C}^{s,\kappa}_t[\Gamma';\Delta' \vdash \mathbf{M}']$$

Proposition 57. *Let* s, κ *be a normal trace, where* $\kappa \in \{\mathsf{fcall}, \mathsf{acall}\}$*, and let* $\Gamma;\Delta \vdash \mathbf{M}$ *be an LTS state in which* signal *does not occur. (a) If* $\Gamma;\Delta \vdash \mathbf{M} \xrightarrow{\kappa}\!\!\!\!\rightarrow$ *then* $\mathbb{C}^s_\kappa[\Gamma;\Delta \vdash \mathbf{M}]\!\downarrow$. *(b) If* $\Gamma;\Delta \vdash \mathbf{M} \not\xrightarrow{\kappa}\!\!\!\!\rightarrow$ *then* $\neg(\mathbb{C}^s_\kappa[\Gamma;\Delta \vdash \mathbf{M}]\!\downarrow)$.

Starting from Definition 20, completeness follows by induction on the length of trace s from Proposition 55, using Propositions 56 and 57.

In rest of this appendix, we describe the strategy for building contexts to satisfy the requirements of Propositions 56 and 57 (up to a structural equivalence that allows reordering of unrelated declarations). Since we are concerned only with normal traces, we adopt the following abbreviations,

$$\begin{aligned} \mathsf{getapp}\ i\ \phi &\triangleq \mathsf{get}\ i, \mathsf{app}\ \phi \\ \mathsf{fcallput}\ \phi &\triangleq \mathsf{fcall}\ \phi, \mathsf{put} \\ \mathsf{acallput}\ \alpha &\triangleq \mathsf{acall}\ \alpha, \mathsf{put} \end{aligned}$$

with completed normal traces formed by the following grammar.

$$\begin{aligned} \textsc{start} &::= \textsc{term}^*, \mathsf{put}, \textsc{ctxt}^* \\ \textsc{term} &::= \mathsf{fcallput}\ \phi, \textsc{ctxt}^*, \mathsf{ret}\ \psi \mid \mathsf{acallput}\ \alpha, \textsc{ctxt}^*, \mathsf{ret}\ \psi \\ \textsc{ctxt} &::= \mathsf{getapp}\ i\ \phi, \textsc{term}^*, \mathsf{put} \mid \mathsf{fun}\ f @ p = \phi \mid \mathsf{adv}\ p = \alpha \end{aligned}$$

We divide labels into three groups: *return labels* (ret), *call labels* (fcallput and acallput), and *context labels* (getapp, put, fun, and adv). A call label is *unreturned* in a normal trace s if the matching ret is not included in s (due to truncation); similarly, a getapp label is *unreturned* if the matching put is missing. A return label is *uncalled* in a suffix of a normal trace if the matching call label is not included.

Recalling Definition 38, $Z(\vec{\mathscr{E}})[M]$ is the term created by iteratively expanding the context stack $\vec{\mathscr{E}}$ and filling the hole of the resulting context with M.

Fix s, t, Γ, Δ, and $\mathbf{M} = \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$. We show how to build term

$$\mathbb{C}_t^s[\Gamma;\Delta \vdash \mathbf{M}] = \mathscr{C}[Z(\vec{\mathscr{H}})[M]].$$

We refer to as $\mathscr{C}$ as *the ambient context*, which is defined below. We refer to the list of contexts $\vec{\mathscr{H}}$ as *the stack*. The stack $\vec{\mathscr{H}}$ alternates contexts from the *term stack* $\vec{\mathscr{E}}$, which is given by the state of the LTS, with those from a *context stack* $\vec{\mathscr{F}}$, which is defined below.

We need to only consider the case where $|\vec{\mathscr{E}}|$ is greater than the number of unreturned call labels in s which have returns in t; otherwise, the final return in t could not occur (by the definition of the LTS). By construction, $|\vec{\mathscr{F}}|$ is the number of unreturned calls in s.

We now define the ambient context $\mathscr{C}$. The ambient context includes function declarations for each name in $\Gamma \cup dn(s,t)$ as well as the declarations Δ and $\vec{A}$. The ambient context also includes the following mutable structures.

- The vector values keeps track of the stored values in a configuration. $put(\mathsf{values}, V)$ pushes V onto the end of the vector; $get(\mathsf{values}, i)$ returns the i^{th} value from the vector; these functions have standard encodings in the lambda calculus with references. In $\mathbb{C}[\_/\_/\_/U_1,\ldots,U_n]$, $get(\mathsf{values}, i)$ returns U_i.
- The reference callcount holds the number of call labels occurring in t; thus, the value of !callcount is 0 in $\mathbb{C}_\varepsilon^s[\_]$.

The functions for $\Gamma \cup dn(s,t)$ are unadvisable, i.e., declared at a fresh primitive pointcut. (Symbolic advice and functions are treated similarly; to simplify the presentation, we abuse notation to allow function declarations at symbolic advice names.) The basic structure of a function body is a case structure on callcount.

$$\begin{aligned}
&\mathsf{fun}\ \phi = \lambda \mathsf{x}.\, \mathsf{callcount}\text{--}\,;\, put(\mathsf{values}, \mathsf{x})\,;\\
&\qquad\qquad \mathsf{case}\ !\mathsf{callcount}\ \mathsf{of}\ \cdots\ \mathsf{default} \Rightarrow \Omega\\
&\mathsf{fun}\ \alpha = \lambda \mathsf{z}.\lambda \mathsf{x}.\, \mathsf{callcount}\text{--}\,;\, put(\mathsf{values}, \mathsf{z})\,;\, put(\mathsf{values}, \mathsf{x})\,;\\
&\qquad\qquad \mathsf{case}\ !\mathsf{callcount}\ \mathsf{of}\ \cdots\ \mathsf{default} \Rightarrow \Omega
\end{aligned}$$

We generate additional cases for these function bodies by working through the trace s,t. The context stack $\vec{\mathscr{F}}$, mentioned above, contains "suffixes" of these function bodies that have been called (in s) but have not yet returned (reading s forward); the context stack includes the actions yet to be performed by these functions (generated by analyzing t). The term stack $\vec{\mathscr{E}}$ includes the suffixes of functions interrupted by a call label; the context stack $\vec{\mathscr{F}}$ includes the suffixes of functions interrupted by a getapp. Call labels that do not have matching returns in s,t will end in Ω, both in the function declaration and in the context stack.

The last element of the trace is treated specially, as initialization. From Proposition 57, we can assume that the last element is a call label. Suppose it is fcallput ϕ (acallput is similar). Then, we add case "$0 \Rightarrow$ signal ()" to the definition of ϕ, and the definition becomes

$$\begin{aligned}&\mathsf{fun}\ \phi = \lambda \mathsf{x}.\, \mathsf{callcount}\text{--}\,;\, \mathit{put}(\mathsf{values}, \mathsf{x})\,;\\ &\qquad\qquad \mathsf{case}\ !\mathsf{callcount}\ \mathsf{of}\ \cdots\ 0 \Rightarrow \mathsf{signal}\ ()\,;\, \mathsf{default} \Rightarrow \Omega.\end{aligned}$$

Now that we have initialized the function declaration, we can begin to generate new cases by working *backwards* through s,t. We generate these using a stack of contexts, called the *generating stack*. When we reach the beginning of t (before getting to the end of s), we record the generating stack, which becomes the context stack $\vec{\mathscr{F}}$. We continue the backwards processing of s to generate the remaining function body cases; this continued processing is performed only to guarantee that function bodies do not change from $\mathbb{C}^{s}_{\kappa,t}[\text{–}]$ to $\mathbb{C}^{s,\kappa}_{t}[\text{–}]$.

Initially the generating stack contains a context "$[\text{–}]\,;\,\Omega$" for every unreturned call label in s,t. Labels are processed as follows:

Return Labels: We push a new context "$[\text{–}]\,;\,\psi$" onto the generating stack for every label ret ψ that we process.

Call Labels: We pop a context $\mathscr{G}$ and add a case "$\Rightarrow \mathscr{G}[()]$" to ϕ for every label fcallput ϕ, and similarly for acallput α. The guard on the case is derived by counting the number of call labels that have been processed.

Context Labels: As we process context labels, we replace the top context of the generating stack $\mathscr{G}$ with a new one, as dictated by the following table. (We use the name y for the variable holding the return value from all calls to term functions; using a single variable name simplifies code generation.)

getapp i ϕ	let y = $(\mathit{get}(\mathsf{values}, i))$ $\phi\,;\,\mathscr{G}$
put	$\mathit{put}(\mathsf{values}, \mathsf{y})\,;\,\mathscr{G}$
fun f@$p = \phi$	fun f@$p = \phi\,;\,\mathscr{G}$
adv $p = \alpha$	adv $p = \alpha\,;\,\mathscr{G}$

This strategy generates contexts that satisfy the requirements. We elide further details.

Editorial for Special Section on Dependencies and Interactions with Aspects

As the use of aspects spreads, it is becoming common to weave multiple aspects into a system, treating different concerns. In this special section, we present three papers that deal with the issue of how aspects may interact, and in particular how they may interfere with each other. Aspect interactions can arise at all stages of software development, including requirements, design, and implementation. The issues somewhat differ at each stage, and in fact for interference itself several definitions are in use.

Interference is sometimes connected to multiple aspects being applied at the same joinpoint, especially when a fixed ordering is not determined using program directives. In that case its detection coincides with determining whether there is overlap in the definitions of pointcuts for different aspects. Another possible definition, seen in the first paper described later, is that one aspect changes the set of joinpoints of another, either adding new ones or deleting ones that previously were in the system. Yet, another type of interference involves name and type conflicts in introductions of fields or methods from different aspects. All of these definitions have the advantage of not requiring specifications of the aspects. That is, it is not necessary to know the intended effect of the aspect.

However, interference can also arise between aspects that do not have common joinpoints, or even common variables. If the intent of the aspects is known, then interference could be defined as a contradiction between the requirements of one aspect and those of another. This could arise already at the level of natural language requirements or when formalizing them into specifications in a logical formalism. The most general semantic definition of interference is that one aspect prevents another from fulfilling its specification, even though each aspect alone woven into a system is correct. Under this definition, even if the requirements of two aspects are in no way contradictory, their implementations may use and modify shared data in a way that one causes the other to operate incorrectly.

This multitude of possible definitions is reflected in the papers in the special section, which each treat somewhat different types of interference.

In the paper *Detection and Resolution of Weaving Interactions* by Günter Kniesel, weaving strategies are shown to influence interaction and interference among aspects, according to the definition above where joinpoints are added or removed. A methodology is given to detect problematic weavings, and to resolve interferences that arise from the weaving strategies themselves. Precise definitions of all of the terms are given, using a first-order logic notation.

The paper *AspectOptima: A Case Study on Aspect Dependencies and Interactions*, by Jörg Kienzle, Ekwa Duala-Ekoko, and Samuel Gélineau, presents a collection of aspects that can be combined in various ways to form different implementations of a transaction manager. The possible conflicts and dependencies provide illuminating

A. Rashid and H. Ossher (Eds.): Transactions on AOSD V, LNCS 5490, pp. 133–134, 2009.

examples of aspect interactions and interferences. These further demonstrate the possible interferences among aspects at the code level that can arise in realistic systems or libraries of aspects and are recommended as a testbed for approaches to aspect interference.

Finally, the paper *Formal Analysis of Aspect-Oriented Models*, by Katharina Mehner, Mattia Monga, and Gabriele Taentzer, deals with interferences among aspects in the requirements stage, providing a formal analysis using a graph transformation analysis tool. Here, of course, only the requirements of each aspect are available, so the possible interferences relate to one preventing another from fulfilling its requirements.

The articles presented here represent a subset of the work currently undertaken on aspects and interactions. However, this is a very young and dynamic field, with a number of most fundamental questions still waiting to be resolved. For instance, as noted above, the community in this field has not even yet agreed on the definition of an aspectual interference/interaction, or whether aspects do cause new types of interference and interaction at all. On the one hand, if, upon weaving, the aspect is dispersed in the modules of the extended paradigm (e.g., in OO classes), then reasoning used to resolve OO-specific interference should suffice to resolve also aspect interaction problems. On the other hand, since aspects provide additional abstraction and composition mechanisms, new reasoning elements reflecting the characteristics of aspects may be needed. This and other equally important questions must be discussed and resolved in order to establish better understanding and acceptance of AOSD. We hope to foster this discussion through this special section as well as with a series of workshops on Aspects, Dependencies, and Interactions held annually with the ECOOP conference.

Finally, we would like to thank all of the authors and referees who have helped in the preparation of this special section. The following three papers provide a cross section of possible approaches to interference, and we hope that they will stimulate additional interest and work on this vital topic.

Ruzanna Chitchyan, Johan Fabry, Shmuel Katz, Arend Rensink
Guest co-editors

Detection and Resolution of Weaving Interactions

Günter Kniesel

Universität Bonn
Institut für Informatik III
Römerstr. 164
D-53117 Bonn, Germany
gk@cs.uni-bonn.de
http://www.cs.uni-bonn.de/~gk

Abstract. Jointly deployed aspects may interact with each other. While some interactions might be intended, unintended interactions (interferences) can break a program. Detecting and resolving interferences is particularly hard if aspects are developed independently, without knowledge of each other.

Work on interference detection has focused so far on the correctness of weaved programs. In this paper, we focus on the *correctness and completeness of aspect weaving*. We show that a large class of interferences result from incorrect or incomplete weaving and present a language-independent analysis of correctness and completeness of weaving.

For certain types of interactions automatic resolution is possible. In this case, our algorithm computes a "weaving schedule" that ensures correctness and completeness of the weaving process. This is possible without special purpose program annotations or formal specifications of aspect semantics. Our technique can check weaving interferences independently of any base program and is applicable to aspects that contain implicit mutual dependencies in their implementation.

1 Introduction

"*Power is nothing without control.*"

Aspect-oriented software development (AOSD) is a powerful new paradigm for separation of concerns. Filman and Friedman [1] characterize AOSD as a modularization concept with two desirable properties: *quantification*, the ability to declare changes to be applied consistently to many places of a program, and *obliviousness*, which means that the extended entities do not need to be aware of being extended and do not need to provide special hooks for enabling extension.

Obliviousness significantly eases unanticipated software evolution and is therefore highly desirable. On the flip side, it makes it hard to compose *independently*

A. Rashid and H. Ossher (Eds.): Transactions on AOSD V, LNCS 5490, pp. 135–186, 2009.

developed software modules without breaking implicit assumptions that they make about their environment [2–4]. This dilemma is known in the component community as the *independent extensibility problem* [5]. In the context of AOSD, unfortunately, the problem is much harder since aspects can impact code much more strongly than component composition does. Component composition must only guarantee that the interaction of a component with its environment adheres to the participating components' expectations. Since aspects can change the internal logic of existing components, they additionally risk breaking the assumptions that a component makes about itself.

Different facets of this specific independent extensibility problem of AOSD have been described as *conflicts* [6–8], *feature interactions* [9–11], *aspect interactions* [6, 7, 9, 12–15], or *aspect interferences* [6, 16–20].[1] The severity of the problem is directly reflected in the considerable attention that it receives in AOSD literature. Many authors consider aspect understandability and verification as one of the main challenges of AOSD [6–9, 12–26].

1.1 Aims

In this paper, we address the problem of unanticipated *joint* deployment of *independently* developed aspects. Our aim is to achieve a solution that

- does not compromise obliviousness in any way,
- is applicable to *black box* aspects, possibly provided by a *third party*,
- does not require programmers to annotate their programs or write semantic specifications of program behavior (neither for the base nor for the aspect parts),
- is independent of the aspect language and base language and
- is independent of the programs to which the analyzed aspects are applied.

Note that unanticipated composition of black box aspects means that language constructs for specifying an explicit ordering of aspects are not sufficient. Because of *unanticipated* composition neither of the jointly deployed aspects can already contain such a specification. Because we want to be able to treat aspects as *black box* components, we cannot assume that the programmers that deploy the third-party aspects know enough of their internal working to determine such an order. Thus, an automated analysis and computation of the "right" order is required.

Independence of base programs means that the analysis should be able to prove that a given set of aspects will not interfere on *any* base program and otherwise should identify *potential* interactions or interferences. In addition, it should be able to decide whether potential interactions actually occur on a *particular* base program by including that base program into the analysis. However, the application of a technique that is dependent on the base program must not be a prerequisite, but just an option for increasing the precision of the analysis.

[1] There is no consistent use of these terms. Some authors distinguish them, others use them interchangeably. We introduce the difference informally in Sect. 1.3 and provide formal definitions as we elaborate our approach (see Definitions 5 and 23).

1.2 Contributions

Our work achieves all the above aims in the context of weaving interactions and interferences (as opposed to semantic interactions within woven programs). In addition, it provides the following novel contributions to the state of the art:

Identification of weaving interactions and interferences. *Weaving interactions* and *weaving interferences* are identified as basic problems that have a wide ranging and often subtle impact on woven programs but have not been recognized before.

Formal foundations. The difference between interactions and interferences is explained by precise, formal definitions for both concepts.

Weaving interaction detection. An algorithm for the fully automated detection of weaving interactions is presented.

Problem category diagnosis. Based on a graph representation of all detected interactions, an analysis is presented that is able to identify subgraphs that represent different problem categories and to determine the required treatment.

Static resolution of weaving interactions. For acyclic parts of the interaction graph, we present an algorithm for computing the "right" weaving order.

Dynamic resolution of weaving interactions. We show that some types of cyclic subgraphs can be treated by "iterative weavers" that are able to perform at run time certain corrective actions.

Weaving interference detection and diagnosis. The proposed analysis detects all interferences, that is, all cyclic subgraphs for which complete and correct weaving is definitely not possible (*conflict*) or cannot be guaranteed without additional information (*ambiguous program*). The algorithm diagnoses and reports the precise cause of the interference.

Interleaved weaving. We show that problems that cannot be treated at the level of aspects can be solved at a finer-grained level. We identify the proper granularity for weaving and introduce the concept of interleaved aspect weaving.

1.3 Approach Overview

We started from a declarative definition of *weaving constraints* that express what we expect from a weaver: structurally complete and correct woven programs. These guarantees are fundamental in the sense that any higher-level analysis of semantic interferences is prone to fail or produce false diagnostics if applied to a program that differs from the one that the programmer expects as a result of weaving.

The definition of correct and complete weaving tells us *what* is expected from a weaver, but not yet *how* to achieve it. To derive an algorithm that enforces correctness and completeness or otherwise diagnoses the cause of errors, it is necessary to understand the mechanisms that collaborate in producing errors: weaving candidates, their selections, weaving interactions, and weaving interferences.

Weaving candidates are aspect language elements at the granularity level of individual introductions and advices that should be woven jointly. A discussion and analysis at the level of complete aspects would be too coarse grained, failing to explain the mechanics of weaving and to resolve problems that can be resolved at the level of individual candidates.

Selections generalize the notion of selected joinpoints. An element of a selection is a tuple consisting of a selected joinpoint together with the context information justifying its selection or is necessary for performing the proper action at that joinpoint. This is necessary to capture also generic aspect languages, which are able to perform different effects at different selected joinpoints, depending on each joinpoint's context [27]. In addition, it helps understand that modifying any part of this information might result in not selecting this tuple any more or in selecting another tuple. This is important for understanding how candidates interact.

Weaving interactions arise if weaving of a candidate adds, deletes, or modifies program elements thus modifying the (future or past) selections of other candidates. The candidate that performs such actions is called the *affecting* candidate, the one whose selection is changed (extended or reduced) is called the *affected* candidate.

Weaving interferences arise if the candidate affected by a weaving interaction is executed before the affecting one. This change of a past selection invalidates the assumptions about the program made by the affected candidate. The invalidated assumptions can be positive (that is, it was assumed that a particular program element exists) or negative (if it was assumed that an element does not exist).

Both types of assumptions might have been the basis for performing or not performing certain actions. Therefore, invalidated assumptions require *corrective actions* to be performed for the affected candidates. If the selection is extended, correction consists of applying their actions at the additional joinpoints. If the selection is reduced, correction undoes actions of the affected candidate at the corresponding joinpoints. A weaver that is able to perform corrective actions is said to support *repair* of interferences.

Weaving errors are caused by weaving interferences that are not repaired. The possible weaving errors are *erroneous effects* applied at the wrong joinpoints and *missing effects*, not applied where they should have.

Unfortunately, existing weavers do not support repair. It is therefore essential to *prevent* interferences. This is possible by detecting interactions in advance and scheduling the execution of candidates such that affected ones are always executed after all those that affect them.

Going beyond our novel problem analysis, we present algorithms for the detection of weaving interactions, prevention of interferences by "proper" ordering and if prevention is not possible, detection of weaving interferences and precise diagnosis of their causes. In this paper, we focus on their application to *static joinpoints*, that is to joinpoints that do not refer to the run time state or execution history of a program.

Our approach is based on a representation of programs as logic fact bases and a representation of weaving candidates as conditional transformations of fact bases. *Conditional transformations* (CTs) are a declarative, logic-based, formal model of program transformations [28, 29]. CTs can be implemented efficiently, as demonstrated by their incarnation in JTransformer [30] and the Conditional Transformation Core (CTC) system [31]. The implementation of the uniformly generic [27, 32] aspect language LogicAJ [33, 34] demonstrates the effectiveness of CTs as a formal model for aspects and target language for aspect compilation.

A CT consists of a precondition and a transformation that is executed on the program elements for which the precondition is true. The assumptions of a CT about the transformed program are captured by its precondition. From the precondition and the transformation it is possible to derive the CTs postcondition, the weakest formula that is guaranteed to be true after the execution of the CT. It captures the state of the program after execution of the CT.

Our analysis leverages on this formal basis by showing that a comparison of pre- and postconditions suffices to detect all potential or concrete *weaving interactions*. Any modification of a CT's assumptions by the effects of another CT corresponds to a pair of unifiable literals[2]: one in the precondition of the affected CT and one in the postcondition of the affecting CT. If both unifiable literals are positive or if both are negative, the affecting CT's postcondition contributes to making the affected CT's precondition true on some additional program elements; thus, it potentially *triggers* the second one. Otherwise, the violating CT's postcondition contributes to invalidating the violated CT's precondition on some program elements; thus, it potentially *inhibits* the second one.

The graph of triggering and inhibition relations provides a precise characterization of the interactions of different weaving candidates. We show that acyclic graphs correspond to programs whose interactions can be resolved automatically, preventing interferences. Graphs with cycles define different problem categories depending on the structure of the cycle. The problems range from the need to use weavers that support interference repair (*repairable interference*), to the need for additional user input (*incomplete program*), and, finally, to the impossibility to weave this set of aspects correctly and completely (*conflict*).

1.4 Paper Overview

In Sect. 2, we *illustrate the problem* of incorrect and incomplete weaving on an example. In Sect. 3, we *analyze the problem*, providing informal definitions of the main concepts (weaving correctness and completeness, interaction, and interference) and sketching the two possible solutions (repair and prevention). In Sect. 4, we introduce *conditional program transformations*, which are a logic-based, formal model for weaving candidates. In Sect. 5, we show how to derive

[2] A literal is a positive or negated predicate symbol applied to n argument terms. For instance, if X and Y are logic variables, a and b are constants, and p is a predicate of arity 2, then p(X,Y), p(a,X), $\neg p(a, b)$ and $\neg p(X, Y)$ are examples of literals for p. Unification makes two literals equal. For details see Appendix 11.

their effects on a woven program (even if the program is yet unknown). In Sect. 6, we show how *potential weaving interactions* between CTs can be detected independently of a base program to which they are applied. Section 7 describes the construction of a graph of potential interactions and its use for determining an *interference-free weaving* order or identifying interferences and diagnosing their cause. This section also shows how iterative weaving can be used to treat certain classes of cyclic weaving dependencies and how our analysis can construct "weaving schedules" also for cyclic cases. Section 8 discusses the applicability of our approach to two semantically equivalent weaving techniques: inlining and use of forwarding methods. Merits, limitations, and extensions of the approach are discussed in Sect. 9. The approach is compared to related work in Sect. 10. Section 11 summarizes our results and concludes.

This paper is a revised version of [35]. In particular, we have added the sects. 3, 7, 8, 9, and 10 and have substantially extended all the others.

2 Problem

Because our work is the first that pinpoints weaving interferences, we must first raise the awareness of the problem and explain how it differs from semantic interferences. In this section, we do it on an example, motivating the need for the deeper problem analysis in the next section.

2.1 An Example: Counters and Getters

This section presents an example adapted from [3] that illustrates how incomplete and incorrect weaving can occur in the case of joint application of independently developed *correct* aspects.

Assume that a programmer is responsible for tuning an application. She might want to identify hot spots by counting accesses to variables (or method invocations). This could be implemented by a `Counter` aspect that

- extends *every* class by a specific counter field for each field in that class and
- adds code to increment the associated counter prior to each direct read access to a field that is not itself a counter field.

Assume further, that another programmer is responsible for ensuring thread safety. She might decide that her task will be easier to fulfill if she could rely on the invariant that every variable access is via accessor methods that can be synchronized. She might thus decide to implement an `Getter` aspect that

- extends *every* class by access methods for each of its fields and
- replaces all direct accesses to these fields by calls of the respective access methods (except for direct read accesses within the access methods themselves).

These are two simple aspects that would typically be developed independently since thread safety and optimization are conceptually independent concerns. Still

they often occur jointly in the same application. It would then be natural to expect that the available aspects can be used together, even if deployed by a programmer that did not develop any of them. This is the scenario that serves as the running example in this paper.

2.2 Waving Interferences in AspectJ

AspectJ and other aspect languages that only support wildcards instead of explicit metavariables can only express instances of the counter-getter example specialized on *particular*, statically known fields. This section demonstrates that weaving problems occur even in such a restricted setting.

Figure 1a shows the base program used for testing. It consists only of the class `Base` and does nothing except that it accesses the field `f`. The class `Base` is the common joinpoint for the introductions, and the access to `f` is the common joinpoint for the advices of the `Counter` and `Getter` aspect.

Figure 1b shows the `Counter` aspect. It introduces the counter `f_count` for the field `f` to the class `Base` and increments it before each access to `f`. A user of this aspect would expect that the counter counts one field access in the above program.

Figure 1c shows the `Getter` aspect. It introduces the getter method `getf` for the field `f` to the class `Base` and enforces its use instead of direct accesses to `f`. A user of this aspect would expect that its application does not change the observable behavior of the base program or of any other aspect. Indeed, the `Getter` aspect implements a refactoring, that is, a behavior-preserving change.

Considering the combination of the invariants that the two aspects should enforce, the result of their joint application should be that

- each access to `f` is via the accessor method,
- the access to `f` in the accessor method is counted,
- the direct access that was replaced by the accessor method invocation is not counted, since it does not exist anymore.

Accordingly, the counter should be incremented once, as part of the execution of the getter method that is used for the only access to `f`.

Figure 1d shows that after weaving the aspects in the order `Counter`, `Getter` with AspectJ 1.5.2, the counter wrongly counts twice: before the invocation of the getter and in the invocation of the getter.

How could this happen? Obviously, the counter, which has been woven first, has inserted a counting statement before the access to `f` in the `run` method of class `Base` and in the method added by the `Getter`. Then the getter replaced the field access in the run method by the invocation of the counter. However, the already inserted counting statement was not removed, although the joinpoint that justified its insertion does not exist anymore. This is a *weaving interference*.

Weaving interferences result from weaknesses of existing weavers, but not from errors in the woven aspects or base programs. They manifest themselves as *weaving errors*, that is, *erroneous effects* that appear at the wrong joinpoints or *missing effects* that do not appear where they should have.

a) The base program:

```
public class Base {
    1   public static void main(String[] args) { new Base().run(); }

    2   private int f = 0;

    3   void run() {
    4     int temp = f; // An access to f
    5   }
    6 }
```

b) The `Counter` aspect:

```
public aspect Counter {

    1   public int Base.f_count = 0 ;

    2   // Count direct accesses to 'f'
    3   before(Base t) :
    4       get(int Base.f) && target(t) {
    5     t.f_count++;
    6     System.out.println(
    7        "Set f_count to " + t.f_count
    8     );
    9   }
   10 }
```

c) The `Getter` aspect:

```
public privileged aspect Getter {

    1   public int Base.getf() {
    2     System.out.println("In getter");
    3     return f;
    4   }

    5   // Enforce use of getter
    6   int around(Base t) :
    7       get(int Base.f) && target(t) {
    8     return t.getf();
    9   }
   10 }
```

d) Result of weaving order `Counter`, `Getter` for AspectJ 1.5.2:

```
Set f_count to 1
   1 In getter
   2 Set f_count to 2
```

e) Result of weaving order `Getter`, `Counter` for AspectJ 1.5.2:

```
In getter
   1 Set f_count to 1
```

f) Result of weaving order `Counter`, `Getter` for AspectJ 1.5.3 to 1.6.3:

```
Set f_count to 1
   1 In getter
```

g) Result of weaving order `Getter`, `Counter` for AspectJ 1.5.3 to 1.6.3:

```
In getter
```

Fig. 1. Weaving interferences in AspectJ: (d) Double counting, (e) Correct behavior: Counting in the getter only, (f) Counting at the wrong place but not in the getter, (g) No counting at all. The full code of the figure can be downloaded from http://roots.iai.uni-bonn.de/research/condor/downloads/pub/Demo_AJ_WeavingInterference.zip

The erroneous effect illustrated in Fig. 1d for AspectJ 1.5.2 also appears in Fig. 1f for AspectJ 1.5.3 to 1.6.3. All versions exhibit the same weaving error if the two aspects are woven in the order `Counter`, `Getter`. Note that in the scenarios where `Getter` is executed first (Fig. 1e, g), `Counter` is activated at a time when the field access joinpoint in `run` has already disappeared. Thus, the counting statement is not added in the wrong place.

The difference between AspectJ versions up to 1.5.2, an versions newer than 1.5.3 (the newest verified version was 1.6.3), is that the `Counter`'s advice is not

applied to the getter method, irrespective of the aspect precedence (see Fig. 1f, g). This appears to be a bug.[3] Therefore, we disregard in the following version 1.5.3 and newer releases up to 1.6.3, which exhibit the same bug.

We conclude that weaving interferences occur in AspectJ, raising a few interesting questions:

- Will interferences occur in any weaver or are they specific to AspectJ?
- How can we distinguish weaving interferences from plain bugs, such as the one discussed above?
- Can we prove that the behavior illustrated in Fig. 1e is the "correct" one?
- Can we determine automatically that the order `Counter`, `Getter` leads to problems whereas `Counter`, `Getter` yields the correct behavior?
- Would a more powerful language than AspectJ exhibit less or more interferences?

2.3 General Weaving Interferences

This section shows that on more complex examples additional weaving errors can occur and that the problem of weaving errors is independent of AspectJ.

Instead of extending our example, we simply analyze it in its full generality introduced in Sect. 2.1, that is we consider what happens if the `Counter` counts and the `Getter` encapsulates *all* field accesses in a program. Corresponding versions of the two aspects can be expressed in *uniformly generic* aspect languages [27, 32] such as LogicAJ [33, 34] or Sally [36]. These languages replace wildcards by explicit metavariables that can be uniformly used in pointcuts, introductions, and advice as place holders for (almost) arbitrary base program elements. Thus, they go beyond type genericity and can express heterogeneous context-dependent actions (e.g., the creation of a different getter method for each field, with field-dependent result type and body).

In order not to tie our discussion to a particular aspect language, we present the generic aspects just via their effects, described informally in Sect. 2.1 and illustrated in Fig. 2.

To show first that incorrect or incomplete weaving does not mean that the aspects are buggy or that the weaving engine is buggy, the behavior of each aspect when woven separately is illustrated in Fig. 2c, d. Part (c) shows the result of weaving only the `Counter` aspect and (d) the result of weaving only the `Getter` aspect. These figures show that each aspect is semantically correct and that the weaver is correct and complete when weaving the aspects *separately*. Still, the remainder of this section shows that weaving errors will occur if the aspects are woven *together*.

Missing Effects. Figure 2c shows that weaving of `Counter` adds the definition of `b_counter` in line 3 and the access to `b_counter` in line 5, which are both candidates for being encapsulated by `Accessor`. Similarly, `Accessor` adds a direct

[3] Thanks to Andrew Clement for his quick confirmation that this is not an intended behavior.

(G) marks lines with effects of `Getter`
(C) marks lines with effects of `Counter`
(C,G) marks lines with cumulated effects of weaving `Counter` and `Getter` (in this order).
Underlining / boxing marks relevant join points added / modified by the last woven aspect.

a) Legend

```
1       public class C {
2           public B b = new B();
3
4           public void useB() {
5               b.doSth();
6           }
7       }
```

b) Initial program

```
1       public class C {
2           public B b = new B();
3   (C)     private int b_cnt = 0;
4
5           public void useB() {
6   (C)         b_cnt = b_cnt+1;
7               b.doSth();
8           }
9       }
```

c) Result of weaving `Counter`.

```
1       public class C {
2           public B b = new B();
3
4   (G)     public B getB() {
5   (G)         return b;
6   (G)     }
7
8           public void useB() {
9   (G)         getB().doSth();
10          }
11      }
```

d) Result of weaving `Getter`.

```
1       public class C {
2           public B b = new B();
3   (C)     private int b_cnt = 0;
4
5   (G)     public int getB_counter() {
6   (G)         return b_cnt;
7   (G)     }
8
9   (G)     public B getB() {
10  (G)         return b;
11  (G)     }
12
13          public void useB() {
14  (C,G)       b_cnt = getB_counter()+1;
15  (G)         getB().doSth();
16          }
17      }
```

e) Result of weaving first `Counter`, then `Getter`: Counting happens in line 11 (where there is no subsequent access to b) and is missing in line 8.

```
1       public class C {
2           public B b = new B();
3   (C)     private int b_cnt = 0;
4
5   (G)     public B getB() {
6   (C)         b_cnt = b_cnt+1;
7   (G)         return b;
8   (G)     }
9
10          public void useB() {
11  (G)         getB().doSth();
12          }
13      }
```

f) Result of weaving first `Getter`, then `Counter`: the access to the counter in line 5 is not encapsulated.

Fig. 2. Result of weaving generic versions of `Counter` and `Getter` either separately (c, d) or jointly (e, f)

access to b in line 4 of (d), which must be counted by `Counter`. Obviously, each aspect adds joinpoints (marked by underlining) that are relevant for the other one when the aspects are woven together.

The Fig. 2e, f presents the result of weaving `Counter` and `Getter` resp. `Getter` and `Counter`, in this order. It shows that the aspect that comes second processes the joinpoints from the original program *and* the joinpoints added by the first aspect. In addition, the respective second aspect creates new joinpoints for the first aspect. In (e), `Getter` adds the `getB` method that contains a yet uncounted access to the variable `b` in line 8. In (f), `Counter` adds line 5, which contains a not-yet encapsulated variable access.

Thus, the Fig. 2e, f demonstrates that, no matter in which order we apply the two aspects in our example, we will miss some effects. The invariant that all field accesses should be counted is violated in (e) and the invariant that all field accesses should be through accessor methods is violated in (f). Depending on the weaving order, either thread safety breaks optimization or optimization compromises thread safety.

Erroneous Effects. Case (e) illustrates that the erroneous effect that occurs in the AspectJ case also occurs with the generic version of the example: An aspect effect (the counter increment in line 11) occurs in a place where there is no suitable joinpoint any more. The increment was added by `Counter` based on the existence of the access to the field `b` in line 6 of Fig. 2c. This field access was removed in step (e) when `Getter` replaced it by an invocation of `getB` (see box in line 12).

Scope of Weaving Interferences. So far we have demonstrated that weaving interferences occur independent of the employed aspect language and affect AspectJ-like languages as well as uniformly generic languages. Note that the problems of missing effects and mutual dependencies also occur in nongeneric languages. With respect to weaving interferences, the only differences between generic and nongeneric aspect languages are that generic languages provide more opportunities for interferences and their aspects contain less hard-coded information about a particular base program, making interference detection that is independent of any base program more difficult.

2.4 Malignity of Weaving Interferences

In general, weaving interferences can

- cause semantic problems,
- undermine nonfunctional properties,
- violate structural assumptions about a program in the sense that program elements are missing or in the wrong place.

Semantic problems are the least harmful ones. They are most likely detected first, for instance, by failed unit tests. However, the causes of such problems are

difficult to trace since neither of the participating aspects can be blamed, each being correct on its own.

If nonfunctional properties are undermined, the problem will most likely pass unnoticed for a while. It might surface much later and in circumstances that might make it very hard to trace the cause back to a missed or erroneous joinpoint. For instance, program shown in Fig. 2f behaves correctly from a purely functional point of view. Nevertheless, the incomplete weaving in line 5 of (f) compromises the thread safety of the program, since the access to the b_counter field is not synchronized.

The third category is the nastiest because it only manifests itself indirectly, during weaving of further aspects. For instance, in case (e), we get the intended functional behavior and the intended thread safety. Still, this apparently correct behavior is the result of the interplay of two different errors, which perfidiously mask each other as long as we only observe the program's behavior but not its structure. The counting of variable accesses does not happen before the variable is accessed (in the getB method), but before the getB method is invoked. This unexpected structure will lead to subtle followup problems, if more aspects are woven. For instance, a program optimization aspect might later be woven dynamically to optimize hot spots where the counters exceed a certain threshold. It will partly fail because it will not find the expected sequences of counter increment and counted variable access. Tracing the cause of this failure back to an apparently correct prior weaving step will be extremely hard.

So far we have introduced the problem of weaving interferences and have explained why their detection and prevention is an important challenge for aspect analysis.

3 Problem Analysis

In this section, we analyze the problem in detail, explain the causes of interferences and sketch two solution approaches. We start by defining informally some language-independent terms for the main aspect concepts relevant for our work. Then, we introduce informally the notion of weaving constraints, weaving interactions, weaving interferences, and conflicts and illustrate these concepts with an example. Formal definitions of these notions are provided in Sect. 6 based on the formal foundations introduced in Sect. 4.

3.1 Aspect Terminology

In the remainder of this paper, we use the following language-independent terminology:

Aspect effect. An aspect effect is a set of changes that an aspect performs on a program. Effects are the addition or deletion of program elements, e.g., the addition of a method to a class or of a statement sequence within a method.

Element. We use the term element to refer to any element of a program.

Aspect predicate. An aspect predicate is a predicate that selects elements. Predicates correspond, for instance, to the pointcuts of AspectJ [37, 38] and filter conditions of Composition Filters [39–41].

Joinpoint. A joinpoint is an element selected by an aspect predicate.

Effect specification. An *a*spect *e*ffect *s*pecification (AES) associates an aspect predicate to an aspect effect such that the effect is executed on the joinpoints selected by the predicate. Effect specifications correspond, for instance, to advice and intertype declarations in AspectJ, filter types in Composition Filters, or composition specifications in Hyper/J [42, 43].

The predicate of an AES can contain conditions that state that some element must *not* exist. In such a case, we say that the effect is *negatively guarded*. Otherwise, the effect is *positively guarded*.

We distinguish base-language elements and aspect-language elements. When we talk about the *base language*, we mean the sum of base-language elements (e.g., the pure Java part of AspectJ). Correspondingly, the *aspect language* is the sum of aspect language elements (e.g., the filter specification, selection, and superimposition constructs in Compose* [39]). Because it simplifies the discussion, this distinction makes sense even in languages that do not make it explicitly, such as Composition Filters [39], Classpects [44], and Sally [36].

3.2 Interactions and Interferences

Aspects can *interact* with base-level modules or with each other. In general, aspect interactions are nothing bad. On the contrary, interactions are necessary. Proper modularization of multiple concerns into multiple aspects depends on the ability of aspects to interact with each other and with the base program to establish the desired overall behavior.

If not detected and treated properly interactions can give rise to interferences. *Interferences* are interactions that violate specified constraints. Depending on the violated constraints, we distinguish the following classes of interferences:

Weaving interference. An aspect's effects violate generic constraints, that define the semantics of aspect weaving.

Semantic interference. An aspect's effects violate application-level constraints.

This paper focuses on weaving interactions and weaving interferences, which have not been addressed previously. Weaving interferences violate the generic constraints that weaving must be correct and complete. They are fundamental in the sense that analysis of semantic interferences depends on the assumption that weaving constraints are not violated. Enforcement of weaving constraints guarantees a *structurally* correct and complete woven programm, which in turn is a prerequisite for any analysis of *semantic* correctness. If the possibility of incorrect or incomplete weaving is ignored, semantic analyzes will wrongly report semantic problems of the application that are actually problems of the employed weaver.

3.3 Weaving Constraints

In this section, correctness and completeness are defined first for a single AES and then generalized to the case of multiple specifications, which can be part of multiple, possibly independently developed, aspects. The separation allows us to distinguish two causes for incorrectness or incompleteness: Wrong implementation of the weaver's joinpoint matching and effect execution functions, on one hand, and weaving interferences, on the other. We are only interested here in the latter.

Weaving a Single AES. A weaver that must only weave individual AESs must provide the following guarantees for any AES *aes* and any program *P*:

Local completeness. Weaving of *aes* must apply the effect of *aes at all* the joinpoints matched in the original program by the predicate of *aes*. For instance, code added by a before advice for the purpose of counting field accesses must appear before each field access matched by the associated pointcut.

Local correctness. Weaving of *aes* must apply the effect of *aes only at* the joinpoints matched in the original program by the predicate of *aes*. For instance, the effect of the counter advice must not be applied in places of the woven program where there is no following field access that should be counted.

Local incorrectness or incompleteness can only arise from a wrong implementation of one of the core functions of the weaver: either the evaluation of predicates that selects joinpoints or the execution of effects must be incorrect.

In this paper, we are only interested in the cases when *global* weaving errors occur *although* the weaver guarantees *local* correctness and completeness. This scenario characterizes the state of the art of current aspect weavers.[4] It can occur as a consequence of aspect interactions if multiple AESs are woven simultaneously.

Weaving Multiple AESs. In this section, we analyze the case when multiple AESs are woven "simultaneously," without a global ordering imposed by the programmer. This is the typical scenario when independently developed black box aspects whose joint use had not been anticipated are deployed together (see Sect. 1.1).

If multiple AESs are woven "simultaneously" it is natural to require that the expectations of each of them are not violated. Each was written with the intention that it will be applied at *all* the matching joinpoints (completeness) and *only* there (correctness). Because of the universal quantification property of aspect predicates, this intention applies not just to the joinpoints of the original program but also to joinpoints added, deleted, or modified by the weaving

[4] In spite of occasional bug reports, we assume that existing weavers are locally correct and complete.

of other effect specifications. This is consistent with known formal models of aspects. For instance, in the aspect sandbox [45], the need to apply an effect specification also to the result of weaving other effect specifications is expressed by defining weaving as a fix point operator.[5] The definition of completeness and correctness presented in this section makes this explicit.

The notation and terminology introduced in Definition 1 helps explaining the relevant issues precisely.

Definition 1 *(Candidates, original program, intermediate programs, woven program).* *The set $AES = \{1, ..., n\}$ of AESs that are to be woven jointly by a locally correct and complete weaver is called the* candidate set. *Each element is a* candidate.

P_0 denotes the original program into which the candidates should be woven.

P_i, for $i = 1, ..., n$, denotes the intermediate result obtained after weaving the candidate i into P_0 or another intermediate result.

$\Pi = \{P_0, P_1, ...P_n\}$ denotes the set of all intermediate results plus the original program.

The woven program *is the final result of weaving all the candidates. The woven program is one of the intermediate results.*

Note that we cannot say which of the intermediate results is the final result, because we do not know the order in which the candidates are woven. In the above definition, the indices only serve to distinguish the individual candidates and programs—they do not express an order. It is the task of the weaver to determine a (parallel or sequentially ordered) schedule for the execution of the candidates that ensures correctness and completeness.

Depending on the chosen schedule for weaving AES, joinpoints that are selected by a candidate, say x, can be removed by the effects of other candidates. The nonremoved ones are called the *surviving selections* of x. They correspond to the "grand total" of original joinpoints plus joinpoint additions and removals during weaving that is relevant for x.

Definition 2 *(Surviving selections).* *Let aes be a candidate (see Definition 1) and $AES' = AES \setminus \{aes\}$ be the rest of the candidates. The selections of aes that survives AES' in Π are the joinpoints selected by aes in any of the programs $P_i \in \Pi$ and not removed by the effect of any element of AES'. We denote them by* surviving(aes, AES', Π).

The definition ignores joinpoint removal by the *same* effect specification. This is because an effect specification can still be applied, even if it removes the

[5] Some language mechanisms violate this principle, providing ways to exclude application of aspects to joinpoints originating from advice, e.g., the `adviceexecution` pointcut in AspectJ or the silent weaver in [13]. We think that these mechanisms should be generally avoided, since hiding aspect effects from other aspects *provokes* mutual constraint violations. For instance, if we hide the counter's effects from the accessor, counting will always undermine thread safety, since the assumption that all accesses, including those to the counter, are via synchronized accessor methods will necessarily be violated (Sects. 2 and 3).

joinpoint that triggered its application. Examples are around advices that do not call `proceed` in AspectJ [37] and error filters in the Composition Filters approach [39]. We come back to this issue when discussing "self-inhibition" in Sect. 7.3.

With the above terminology, we can define that weaving is complete and correct if and only if each candidate performs its effects precisely at its surviving selections:

Definition 3 *(Constraints: Global completeness and correctness). For any candidate set AES and any initial program P_0 the effect of each aes $\in A$ must be applied*

- at all *the surviving selections of aes (global completeness) and*
- only at *the surviving selections of aes (global correctness).*

The following examples illustrate the implications of the above constraints.

Example 1 (Global completeness). Assume that $AES = \{cnt, add\_acc\}$, *cnt* counts accesses to the field f, and *add_acc* adds a getter method for f.

Then global completeness requires that counting by *cnt* must be performed also for the access to f in the body of the method introduced by *add_acc*. The access to f in the body of the method introduced by *add_acc* is part of the surviving selections of *cnt* because it is part of P_{cnt}, selected in P_{cnt} by *cnt* and clearly not removed by any other candidate (since in this case the only other candidate is *add_acc*, which added the access).

Example 2 (Global correctness). Assume that $AES = \{cnt, use\_acc\}$, *cnt* is the one from Example 1, and *use_acc* replaces field accesses with accessor method invocations.

Then, global correctness requires that counting of accesses to f by *cnt* must *not* be performed in places where the accesses to f have been replaced with accessor method invocations by *use_acc*. The replaced accesses to f are not part of the surviving selections of *cnt* because they have been removed by *use_acc*.

Example 3 (Interplay of global correctness and completeness). Assume that $AES = \{cnt, add\_acc, use\_acc\}$ with *cnt* and *add_acc* defined as in Example 1 and *use_acc* defined as in Example 2.

Then, global completeness requires that counting of accesses to f by *cnt* is performed also in the body of the accessor method added by *add_acc*. Global correctness requires that counting must *not* be performed in places where the accesses to f have been replaced with accessor method invocations by *use_acc*. Thus, the woven program will correctly count the accesses to f.

Note how correctness and completeness play together in to enforce a weaving result whose behavior matches the intuitive expectations of the programmer. Without correctness, we would accept that counting is performed not just in the accessor methods but also at the place of their invocations, resulting in twice as high counter values as correct. Without completeness, we would accept

that counting is neither performed at the place of accessor method invocations nor in the accessor methods; thus, not counting at all. If both, completeness and correctness, would be violated counting would happen at the place of the accessor invocations but not in the accessor methods. This would yield the expected counter value but would produce a program that is structurally incorrect leading to subtle follow-up problems (see Sect. 2.4).

Complete and correct weaving are necessary and sufficient conditions for a complete and *structurally* correct woven programm, which in turn is a prerequisite for any analysis of *semantic* correctness.

3.4 Weaving Errors

The definition of correct and complete weaving[6] tells us declaratively *what* we expect from a weaver, but not *how* to achieve it. To derive an algorithm that enforces correctness and completeness or otherwise diagnoses the cause of errors we need in the first place define the errors and identify their causes.

There are two types of weaving errors corresponding to the two types of violated weaving constraints: incorrect weaving results in *erroneous effects* and incomplete weaving results in *missing effects*.

Definition 4 *(Missing and erroneous effects). Let aes be a candidate (see Definition 1).*

A missing effect of *aes is an effect of aes that has* not *been applied at one of its surviving selections. An* erroneous effect of *aes is an effect of aes that has been applied at a joinpoint that is* not *among the surviving selections of aes.*

3.5 Weaving Interactions

Both kinds of errors are effects that had (or had not) been applied based on assumptions that have been invalidated later. The invalidated assumptions are either the existence or nonexistence of elements. The invalidating actions are the subsequent removal or addition of these elements by the effect of another candidate.

Please recall that the assumptions of a candidate are expressed by its predicate and, more precisely, the set of successful substitutions for the variables of the predicate.[7] Thus, invalidated negative or positive assumptions correspond to additional or lost successful substitutions. This is expressed by the notion of triggering and inhibition introduced in this section. They capture the *weaving interactions* that can occur between candidates.

Definition 5 *(Triggering and Inhibition). The following two kinds of* weaving interactions *can occur among any two jointly woven candidates a and b:*

[6] Henceforth, we always mean "globally correct," etc., although we simply write "correct," for brevity.

[7] Successful substitutions are substitutions that make the predicate true (see also Appendix 11).

Triggering. *The candidate a* triggers *or* enables *b for a substitution θ if it adds, removes, or modifies elements such that b's predicate additionally succeeds for at least one previously false substitution.*

Inhibition. *The candidate a* inhibits *or* disables *b for a substitution θ if it adds, removes, or modifies elements such that b's predicate becomes false for at least one previously successful substitution.*

If a triggers or inhibits b we say that a affects *b and call a the* affecting *and b the* affected *partner of the interaction.*

Positively guarded effect specifications (Sect. 3.1) can be triggered by the addition of an element and inhibited by the deletion of an element. Negatively guarded specifications can be triggered by the deletion of an element and inhibited by the addition of an element.

Example 4 (Triggering). For the weaving candidates described in Example 1, we can observe that *add_acc* triggers *cnt* since it adds an element (an access to the field *f*) that lets the predicate of *cnt* succeed for one additional substitution.

Let us additionally consider the *use_acc* candidate from Example 2. A sensible implementation of *use_acc* will check in its predicate that the methods that it should invoke exist. Thus, *add_acc* also triggers *use_acc* since it adds the methods that are to be invoked by *use_acc* thus making its predicate true.

Example 5 (Inhibition). For the weaving candidates described in Example 2, we can conclude that *use_acc* inhibits *cnt* since it removes elements (accesses to the field *f*) invalidating the predicate of *cnt* for many previously successful substitutions.

3.6 Weaving Interferences

Whether an interaction leads to an interference depends on the order of execution of the two interacting candidates. An interference is an interaction whose affected partner is executed before the affecting one.

Definition 6 *(Interference: Missed trigger and missed inhibition). Let a affect b for the substitution θ. An* interference *occurs if a is executed before b. If a triggers b the interference is called a* missed trigger*. If a inhibits b the interference is called a* missed inhibition*.*

Example 6 (Missed Triggering). Continuing Examples 1 and 4, we can observe that if *cnt* (the affected partner) is executed before *add_acc* (the affecting partner) then *cnt* will fail to notice the field access added by *add_acc*. This will result in a missing effect.

Example 7 (Missed Inhibition). Continuing Examples 2 and 5, we can observe that if *cnt* (the affected partner) is executed before *use_acc* (the affecting partner) then *cnt* will count at all the field accesses that will subsequently be removed by *add_acc*. This will result in many erroneous effects.

3.7 Prevention and Repair

So far we have analyzed the problem, identifying the constraints that we want to enforce, the errors that occur if they are violated, and the possible causes of the errors. Understanding that the causes of errors are missed interactions due to "wrong" execution order of candidates immediately suggests two possible solutions: prevention and repair.

Prevention. Prevention means determining in advance an order that ensures correct and complete weaving. The main idea is to statically analyze the candidate set for *potential* triggering and inhibition relations and to order it such that each candidate is executed after all the candidates whose effects could trigger or inhibit it. Ordering is possible if the weaving interactions build an acyclic graph, which corresponds to a partial order.

Definition 7 *(Required Weaving Order)***.** *Let AES be a candidate set and let a and b be any two elements of AES. The required weaving order is a partial order on the candidates defined as:*

$$a \text{ before } b \equiv a \text{ triggers } b \vee a \text{ inhibits } b$$

Theorem 1 *(Prevention)***.** *Any total order that is consistent with the required weaving order is guaranteed to ensure correctness and completeness of weaving.*

Proof outline: *Any total order that is consistent with the required weaving order guarantees that each candidate x is executed after all the candidates whose effects could trigger or inhibit it. Thus, the effects of all the triggers and inhibitions that affect x are already part of the intermediate result on which x is executed. Hence, x cannot miss any relevant triggers or inhibitions. Thus, weaving errors resulting from weaving interferences cannot occur. Since we assumed a locally correct and complete weaver local errors will not occur either. Absence of* any *weaving errors is equivalent to correctness and completeness. Thus, we have proved the claim.* □

Example 8 (Prevention). Assume the scenario of Example 3 where {*cnt*, *add_acc*, *use_acc*} are to be woven together. Because of their weaving interactions discussed in Examples 4 and 5, these candidates exhibit the following ordering constraints:

- *add_acc* before *use_acc* because *add_acc* triggers *use_acc*,
- *add_acc* before *cnt* because *add_acc* triggers *cnt*,
- *use_acc* before *cnt* because *use_acc* inhibits *cnt*.

Indeed, if the candidates are woven in the order *add_acc*, *use_acc*, *cnt* global completeness and correctness is preserved because *use_acc* cannot miss the accessor method added by *add_acc* and *cnt* can neither miss the field access in the accessor method nor can it add erroneous counting statements (the field accesses have already been removed by *use_acc*).

Unfortunately, the prevention approach is not applicable if the interaction graph is cyclic, hence does not define an order. This scenario would occur if we extended our example by considering that the counting of field accesses by *cnt* needs itself a counter field and an access to this counter field. For completeness, this access must also be performed via an accessor method. Thus, *cnt* triggers *use_acc*. Because *use_acc* inhibits *cnt* we would have a cycle.

In cases when prevention of weaving interferences is not possible, there is still the option to repair interferences.

Repair. Repair means employing a weaver that monitors the weaving process, notices weaving interferences after the fact, and performs the following *corrective actions*:

- *Redo* applies a previous candidate's effect at joinpoint *jp* when elements that trigger the candidate for *jp* are added or elements that inhibited the candidate for *jp* are removed.
- *Undo* rolls back a previous candidate's effect at joinpoint *jp* when elements that inhibit that candidate for *jp* are added or elements that triggered that candidate for *jp* are removed.

Note that in addition to the above *systematic repair*, some interferences can be *repaired accidentally* by further interferences. This happens when other candidates remove or add again some of the triggering or inhibiting elements. This would also be a legal repair, since only the surviving selections are relevant for correctness and completeness (see Definitions 2 and 3). Accidental repair is certainly not a solution for interferences. However, proper understanding of this

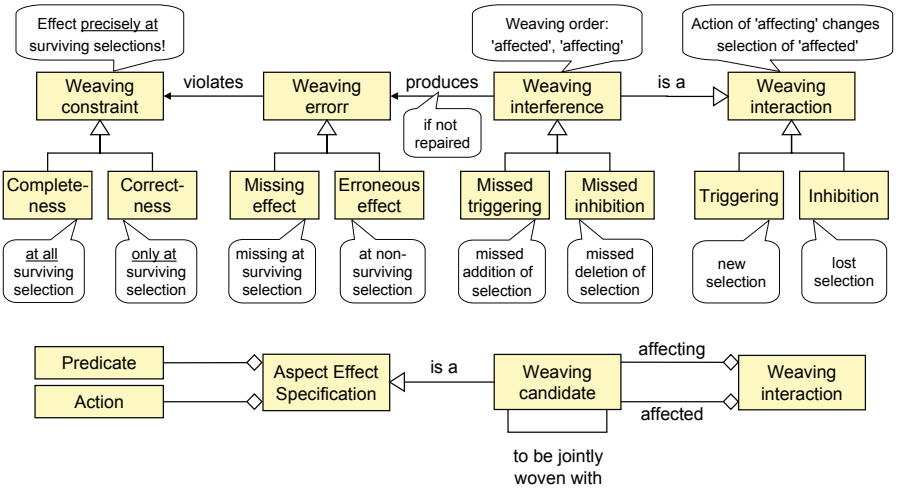


Fig. 3. Summary of problem analysis: weaving constraints are violated by weaving errors that arise when weaving interferences are not repaired. Weaving interferences arise if the weaving candidate affected by a weaving interaction is executed before the affecting one.

issue helps to refine the precision of interference analysis by excluding accidental repair scenarios from the set of reported interferences.

Prevention Versus Repair. Prevention enables use of a simple *linear weaver* that applies every aspect effect only once to all the matching joinpoints. Thus, prevention is compatible with existing weavers. Being a purely static approach, prevention is also efficient since it does not require any additional run time actions.

Repair requires extension of existing weavers into weavers that include monitoring and corrective actions. Because corrective actions might cascade, we call these *cascading weavers*. Cascading weavers require a more complex infrastructure than linear ones. Their weaving process can take significantly longer or not terminate at all in certain cases of cyclic interactions.

So prevention, if applicable, is clearly the preferred strategy. However, prevention depends on the ability to compute in advance a weaving order based on a static analysis of the weaving candidates that identifies their potential interactions. Showing how this can be achieved is the main topic of the remainder of this paper.

3.8 Summary

This section has presented a thorough problem analysis leading to a sketch of two solution approaches, prevention and repair. A summary of the problem domain is expressed as an UML diagram in Fig. 3.

We started from a declarative definition of weaving constraints that express what we expect from a weaver: structurally complete and correct woven programs. Weaving errors are programs that violate these constraints. The possible weaving errors are erroneous and missing effects. Their cause is the occurrence of weaving interferences that had not been repaired. Weaving interferences arise if the weaving candidate affected by a weaving interaction is executed before the affecting one. Executing the interacting partners in the opposite order prevents interferences and errors. Interactions are effects of one weaving candidate that modify the selection of another candidate.

In addition to the problem analysis, this section provided an almost complete overview of our approach at an informal level. The remainder of this paper can provide the formal foundation for the analysis and fully automated solutions to the addressed problems.

4 Conditional Transformations

In this section, we provide the formal foundations for the concepts introduced in the previous section. Table 1 summarizes the correspondence of the informal terms from Sect. 3 to their formal counterparts. We define a logic-based representation of programs and conditional transformations (CTs) on this representation, explain the relation of CTs to aspects, and show how the example introduced

Table 1. Correspondence of informal and formal terms

Informal	Formal
Program element	Logic term
Program	Logic fact base
Aspect predicate	Condition C
Joinpoint	Substitution θ
Selection	Substitution set Θ
Aspect effect	Transformation T
Effect specification	Conditional transformation $CT \equiv C \rightarrow T$
Aspect	Set of conditional transformations

informally in the previous section can be expressed with CTs. CTs are the basis for the weaving interaction analysis in the next section. We only introduce here the parts of the CT concept relevant for this paper. A complete treatment can be found in [29] and [28]. CTs can be implemented efficiently, as demonstrated by their incarnation in JTransformer [30] and the Conditional Transformation Core (CTC) system [31]. The implementation of the uniformly generic [27, 32] aspect language LogicAJ [33, 34, 46] demonstrates the effectiveness of CTs as a formal model for aspects and target language for aspect compilation.

Aspect weaving is a transformation of programs. From the point of view of weaving, an aspect can be interpreted as the specification of *what* is to be transformed and *how* it should be transformed. This can be generalized as the specification of conditions and transformations. Conditions correspond to aspect predicates and transformations correspond to aspect effects (3.1).

4.1 Logic Language

Programs, conditions, and transformations are represented symbolically as atoms and terms in a first-order *logic language* LL consisting of

- the predefined predicates exists($Elem$) (for checking that a program element exists), $A = B$ (unification), and concat(A, B, AB) (concatenation) plus an arbitrary set of programmer-defined, possibly recursive, predicates built on the basis of the predefined ones,
- function symbols representing program element types,
- constants, and
- the logic symbols $\wedge, \vee, \neg, =, \forall$, and $\exists$.[8]

4.2 Base Language: Logic Terms

A *program element* is represented by a variable-free logic term. The function symbol represents the *element type*. The first argument of each term represents the *identity* of the element. All other arguments represent the values of its

[8] For brevity of notation, we suppress quantifiers in formulas. Variables that occur in positive atoms are universally quantified. Variables in negated atoms that do not occur in any positive atom are existentially quantified.

Table 2. Sample term representation of AST elements

class($id, name$)	A class named *name*
field($id, class, name, type$)	A field in *class*
method($id, class, name, type$)	A method in *class*
getfield($id, meth, field$)	An access to *field* in *meth*
call($id, meth, m2$)	Call of $m2$ in *meth*
incr($id, meth, field$)	An increment of *field* in *meth*
return($id, meth, expr$)	A return of *expr* in *meth*

components. Components can be either attributes or edges. An *attribute* has a value from a primitive domain (boolean, numerical, or string). The value of an *edge* is the identity of the element to which the edge points.

A particular choice of function symbols and constants defines a particular base language.

Definition 8 *(Base language).* *The* base language Σ *is the set of all possible terms that can be built from the chosen function symbols and constants. In logic programming terminology, this corresponds to the* Herbrand universe *of the logic language (see Appendix 11).*

Table 2 presents a selection of function symbols that we use in our subsequent examples along with an informal explanation of the meaning of each term. Note, however, that their meaning is relevant only for the translation of a concrete language to our term representation and back again. During the analysis, the terms are treated as symbolic information that is not interpreted; therefore, our analysis is *independent* of the base language.

The representation in Table 2 is only meant to illustrate a principle. For brevity, the used terms are strongly simplified. A complete term representation of the abstract syntax tree (AST) for Java 1.4 is described on the JTransformer website [30]. JTransformer is our Eclipse plugin for converting Java programs into a logic-based representation, applying conditional transformations, converting the result back to Java, and incrementally synchronizing the logic representation with changes of the Java source.

4.3 Base Programs: Logic Facts

The following definition expresses that an original program is represented by a set of ground (variable-free) atoms for the "exists" predicate.

Table 3. Logic fact representation of the program from Fig. 2b

exists(class(c1,'C')).	`// class C`
exists(field(f1,c1,b,'B')).	`// B b;`
exists(method(m1,c1,useB,void)).	`// void useB()`
exists(return(ret1,m1,get1)).	`// return b`
exists(getfield(get1,m1,f1)).	`// b`

Definition 9 *(Program). Let LL_{core} be a version of the logic language LL defined in Sect. 4.1 restricted to contain only the predicate "exists". The set Π of all programs for LL is the power set of the Herbrand base of LL_{core} (see Appendix 11). A* program *is an element of Π.*

A correct and complete representation contains the atom exists(n) if and only if the program element represented by n exists in the original program. Henceforth, we assume correct and complete representations.

Example 9 (Program as logic factbase). The factbase shown in Table 3 represents the program from Fig. 2b using the terms defined in Table 2.

4.4 Pointcut Language: Conditions

Any condition about a program can be expressed by a condition about the existence of particular elements and their properties. To express constraints on the properties of elements, the condition language includes unification ($A = B$), the test for nonunifiability ($A \neq B$), and concatenation (concat(A, B, AB)). The concat predicate is included to express naming patterns commonly used in aspect languages. If the third argument is a free variable it is bound to the concatenation of the first two. If the third argument is nonvariable then it is split, binding the first two arguments to a suitable prefix and postfix. When used with all arguments bound to nonvariable terms, the predicate simply works as a check. *Cond* is the set of all formulas that can be built in the logic language introduced in Sect. 4.2.

Formulas can contain variables, denoted by leading capital letters and italic font, e.g., *Class*. The underscore, _, represents anonymous variables. Different occurrences of _ represent different variables. Constants start with a small letter and are set in roman font, e.g., myField. If constants must start with a capital letter they are enclosed in quotes, e.g., "MyClass".

The following examples show how some of the conditions relevant for our counter-getter example from Sect. 2 can be expressed in the mini-language defined in Table 2.

Example 10 (Field without getter). Assuming that getter methods have the same type as the respective field, are defined in the same class as the field, and the name of the getter method is the name of the field with the added prefix "get," the following condition selects "all fields that have no getter method":

$$\text{exists}(\text{field}(\_, C, N, T)) \wedge \text{concat}(\text{get}, N, GetN) \wedge \neg\, \text{exists}(\text{method}(\_, C, GetN, T))$$

The next two examples show how the condition language can be extended by own predicates, enabling modularization and reuse of conditions.

Example 11 (Fields without counter). Assuming that counter fields are marked by the suffix "_cnt" added to the name of the counted field, we can represent "all

fields F that are not themselves counter fields and do not have a counter field" by the predicate:

$$
\begin{aligned}
&\text{fieldWithoutCounter}(F, C, CntN) \equiv \\
&\quad \text{exists}(\text{field}(F, C, N, \_)) && // \ F \text{ is a field named } N \\
&\quad \wedge \neg\, \text{concat}(\_, \text{\_cnt}, N) && // \ F \text{ is no counter} \\
&\quad \wedge\, \text{concat}(N, \text{\_cnt}, CntN) && // \ CntN \text{ is the name of } F\text{'s counter} \\
&\quad \wedge \neg\, \text{exists}(\text{field}(Cnt, C, CntN, \text{int})) && // \ F \text{ has no counter}
\end{aligned}
$$

Example 12 (Uncounted field access). "All uncounted read accesses to a field that has a counter field and is not itself a counter field" can be expressed as

$$
\begin{aligned}
&\text{uncountedFieldAccess}(Acc, M, Cnt) \equiv \\
&\quad \text{exists}(\text{getfield}(Acc, M, F)) && // \text{ Method } M \text{ contains an access} \\
&\quad \wedge\, \text{exists}(\text{field}(F, C, N, \_)) && // \text{ ... to the field } F \text{ of class } C \\
&\quad \wedge \neg\, \text{concat}(\_, \text{\_cnt}, N) && // \ F \text{ is no counter} \\
&\quad \wedge\, \text{concat}(N, \text{\_cnt}, CntN) && // \ CntN \text{ is the name of } F\text{'s counter} \\
&\quad \wedge\, \text{exists}(\text{field}(Cnt, C, CntN, \text{int})) && // \ Cnt \text{ is } F\text{'s counter defined in } C \\
&\quad \wedge \neg\, \text{exists}(\text{incr}(\_, M, Cnt)) && // \ Cnt \text{ is not incremented in } M
\end{aligned}
$$

Conditions that contain variables can be true for different substitutions of values for the variables. In the context of aspects, we are typically interested in the set of all the substitutions that make a condition true with respect to a particular program.

Definition 10 *(Substitution Set, Success, Failure).* *We write $P \vDash C\Theta$ to indicate that Θ is the nonempty set of all most general substitutions for variables of C that make C true in the interpretation induced by the program P. If there is no such substitution, we write $P \nvDash C$ and say that C* failed *in P. Otherwise, we say that C is* succeeded.

Note that the empty substitution set differs from the substitution set containing only an empty substitution. The former indicates failure, the later indicates success of a variable-free condition.

Example 13 (Substitutions). Let P be the program from Table 3. Then

- $P \vDash \text{fieldWithoutCounter}(F, C, N)\{[F \leftarrow f1,\ C \leftarrow c1,\ N \leftarrow b\_cnt]\}$
- $P \nvDash \text{uncountedFieldAccess}(A, M, Cnt)$ because P contains no field that has a counter field.

Note that in this example a condition has only one successful substitution and the other one has none. Later we will see how additional substitutions will "appear" as a result of interactions.

4.5 Aspect Effects: Transformations

Our transformation language consists of just four basic operations: adding an element, deleting an element, creating a new element identity, and executing a sequence of such operations[9]:

Definition 11 *(Transformation Language). The following grammar defines the set τ of all well-formed transformation expressions that can be built from Σ (see Sect. 4.2) and the additional function symbols* $\{\text{add}, \text{del}, \text{newId}, \circ\}$.

$$
\begin{array}{lll}
T & \rightarrow ET & \\
 & \mid ET \circ T & \\
ET & \rightarrow \text{add}(elem) & : elem \in \Sigma \\
 & \mid \text{del}(elem) & : elem \in \Sigma \\
 & \mid \text{newId}(var) & : var \text{ is a logic variable}
\end{array}
$$

The newId operation is used to create a new, unique identity (see Sect. 4.2) for each new element added to a program. The transformation engine within an aspect weaver can be defined concisely as the following interpreter that consumes an expression of the transformation language and a program and produces a new program.

Definition 12 *(Transformation Engine). Let n, n_1, n_2 be variable-free atoms from Σ (representing AST nodes), e_1, e_2 be transformation language expressions from τ, and let P be a program. The transformation interpreter is defined by the function* transform*:*

$$
\begin{array}{rcl}
\text{transform} : \tau \times \Pi \longrightarrow \Pi & & \\
\text{transform}(add(n), P) & = & \begin{cases} P \cup \{\text{exists}(n)\} & : \text{exists}(n) \notin P \\ P & : otherwise \end{cases} \\
\text{transform}(del(n), P) & = & \begin{cases} P \setminus \{\text{exists}(n)\} & : \text{exists}(n) \in P \\ P & : otherwise \end{cases} \\
\text{transform}(e_2 \circ e_1, P) & = & \text{transform}(e_2, \text{transform}(e_1, P))
\end{array}
$$

The constraint that the arguments to *add* and *del* must be variable free ensures that the transformation does not produce incomplete programs.

4.6 Effect Specifications: Conditional Transformations

Definition 13 *(Conditional transformation). A* conditional transformation *$CT \equiv c \rightarrow t$ is a pair consisting of a formula $C \in Cond$ and a transformation expression $T \in \tau$ such that every variable V from T that is not contained in C is contained in a newId(V) operation. The formula C is called the* precondition *of the CT.*

[9] For brevity, we omitted the replacement operation described in [47]. It can be represented by deletion and immediate addition.

The constraint that all variables in T must be bound either by the condition or a newId operation ensures that the transformation does not produce incomplete programs.

The following two CTs implement the generic version of the `Counter` aspect introduced in Sect. 2.

Example 14 (AddCnt: Add counter fields). The CT `AddCnt` extends every class by a specific counter field for each field of that class that is not itself a counter field and does not yet have a counter field. Its precondition is defined in Example 11. The transformation just adds the counter field with a new identity:

$$\begin{aligned}&\mathrm{AddCnt}(F, C, Cnt) \equiv \\ &\quad \mathrm{fieldWithoutCounter}(F, C, CntName) \\ &\quad \longrightarrow \\ &\quad \mathrm{add}(\mathrm{field}(Cnt, C, CntName, \mathrm{int})) \circ \mathrm{newId}(Cnt)\end{aligned}$$

Example 15 (UseCnt: Count field accesses). The CT `UseCnt` adds code to increment the associated counter before each read access to a field that is not itself a counter field. The uncountedFieldAccess condition is defined in Example 12.

$$\begin{aligned}&\mathrm{UseCnt}(F, C, Cnt) \equiv \\ &\quad \mathrm{uncountedFieldAccess}(Acc, Meth, Cnt)) \\ &\quad \longrightarrow \\ &\quad \mathrm{add}(\mathrm{incr}(Inc, Meth, Cnt)) \circ \mathrm{newId}(Inc)\end{aligned}$$

The process of determining joinpoints and applying the specified effects at every joinpoint corresponds to the application of a conditional transformation to a program.

Definition 14 *(CT application). Application of a conditional transformation $CT \equiv c \rightarrow t$ to a program P consists in*

1. *determining Θ, the set of all substitution for the variables in C that make C true in P,*
2. *applying each of the computed substitutions to T, and*
3. *executing all the resulting transformations on P.*

Each computed substitution corresponds to a joinpoint together with all the context information necessary to establish validity of the joinpoint and to perform the associated transformation.

The introduction of logic-based conditional transformations given here suffices for the purpose of this paper. For additional information see [28, 29].

4.7 Aspects: Sets of CTs

An aspect is just a set of conditional transformations. For instance, the `Counter` aspect is implemented by the `AddCnt` and `UseCnt` CTs introduced above. The

`getter` aspect also corresponds to a pair of CTs: `GetI` performs changes at interface level (addition of method), and `GetC` performs changes at code level (replacement of field access by method invocation). Their interaction-related parts are shown in Fig. 4.

Sets of CTs can be used as an intermediate representation for the compilation of aspects. For instance, in LogicAJ [33, 34] aspects are compiled to CT sets reducing them to their essence that is relevant to a weaver. This approach has many advantages. First, it simplifies the language implementation by separating the parsing and compilation concern from the weaving concern. Second, it reduces weaving to the execution of a small and highly efficient generic transformation engine. Last, but not least, the logic-based intermediate representation enables various analyzes and optimizations. In this paper, we demonstrate one of them: the ability to determine weaving interactions and derive automatically the order in which the weaver has to process individual CTs.

A set of aspects is the union of the CT sets representing the individual aspects. Hence, ordering CT's from different aspects is the same as ordering the CTs of one aspect. Both require to determine an order between the elements of a CT set. This is the topic of Sect. 6.

5 Derivation of Effects

The CT formalism enables different static analyzes and operations on CTs. In this section, we introduce the calculation of the cumulated effect of a CT, which is the basis for the interaction detection presented in the next section.

The *cumulated effect* of a CT is the weakest formula that is guaranteed to be true after successful application of the CT to *any* program P. "Weakest" indicates that we disregard any conditions that were true when the execution of the CT started. In the following, the calculation of a CT's cummulated effect is split into the following steps:

- calculation of the *effect* of an individual transformation,
- calculation of the *invariant* part of a precondition that is still true after a transformation,
- calculation of the *postcondition* that is true after execution of a single transformation in a state where a certain precondition was true before execution,
- calculation of the *cumulated postcondition* of a transformation sequence for a given precondition, and
- calculation of the *cumulated effect* and *cumulated postcondition* of a CT.

5.1 Elementary Transformation Effects

The effect of adding an element to a program is existence of the element in the transformed program. The effect of deletion is that the deleted element will not exist in the transformed program.

Definition 15 *(Elementary Transformation Effect)*

$$\begin{aligned}
&\text{effect} : T \rightarrow Cond \\
&\text{effect}(\text{add}(elem)) &&= \text{exists}(elem) && : elem \in \Sigma \\
&\text{effect}(\text{delete}(elem)) &&= \neg\,\text{exists}(elem) && : elem \in \Sigma \\
&\text{effect}(\text{newId}(_)) &&= \text{true}
\end{aligned}$$

Example 16 (Elementary Transformation Effect). For $t_1 = \text{del}(\text{field}(F, C, N, \text{int}))$, $\text{effect}(t_1) = \neg\,\text{exists}(\text{field}(F, C, N, \text{int}))$. For $t_2 = \text{add}(\text{method}(M, C, N, \text{void}))$, $\text{effect}(t_2) = \text{exists}(\text{method}(M, C, N, \text{void}))$.

Individual transformation effects are independent of any program that existed or any condition that was true before the transformation was executed. Given a precondition c that was true before the execution of an elementary transformation t, we can calculate the invariant of c and t, that is a condition that expresses what will *still* be true after execution of t.

5.2 Invariants

The invariant of c and t can be derived by considering that execution of t can invalidate previously successful substitutions for some literals of c. The affected literals are those that are unifiable with the negation of t's effect. For each affected literal l and effect e, the invalidated substitutions are instances of $\theta = \text{mgu}(l, \neg e)$, the most general unifier of l and $\neg e$ (see Appendix 11). The invariant can be obtained from c by adding the negation of the unifier conjunctively to the respective affected literal. This addition acts as a filter that excludes the invalidated substitutions.

Definition 16 *(Invariant).* *Let c, c_1, c_2 denote conditions and let e be an elementary transformation effect.*

$$\text{invar} : Cond \times Cond \longrightarrow Cond$$

$$\text{invar}(c_1 \vee c_2, e) = \text{invar}(c_1, e) \vee \text{invar}(c_2, e) \tag{1}$$

$$\text{invar}(c_1 \wedge c_2, e) = \text{invar}(c_1, e) \wedge \text{invar}(c_2, e) \tag{2}$$

$$\text{invar}(c, e)) = c \quad : \neg\exists\theta : \theta = \text{mgu}(c, \neg e) \tag{3}$$

$$\text{invar}(c, e) = c \wedge \neg\,\text{eq}(\theta) : \exists\theta : \theta = \text{mgu}(c, \neg e) \tag{4}$$

The first two equations just express the recursive descent into every part of a condition, down to the level of individual literals. The third equation says that literals that are not unifiable with the negation of the effect remain unchanged. The last equation adds the negation of the unifier θ conjunctively to the affected literal. For this, the invocation $\text{eq}(\theta)$ represents the unifier as the conjunction of unifications that it imposes. This is illustrated in Example 17 below.

Definition 17 *(Unifiability constraints). For each binding $v_i \leftarrow t_i$ in θ, the conjunction contains the constraint $v_i = t_i\theta$:*

$$\mathrm{eq}(\theta) = \begin{cases} v_1 = t_1\theta \wedge \ldots \wedge v_n = t_n\theta \\ \quad : \theta = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\} \\ true : \theta = \{\} \end{cases}$$

Example 17 (Invariant). Let $t = \mathrm{del}(\mathrm{field}(Field, C, Name, \mathrm{int}))$, $c = l_1 \wedge l_2$, $l_1 = \mathrm{exists}(\mathrm{class}(C, CN))$ and $l_2 = \mathrm{exists}(\mathrm{field}(F, C, FN, T))$. Then

$$\begin{aligned} e = \mathrm{effect}(t) &= \neg\,\mathrm{exists}(\mathrm{field}(Field, C, Name, \mathrm{int})) \\ \mathrm{mgu}(l_2, \neg e)) &= \{F \leftarrow Field, FN \leftarrow Name, T \leftarrow \mathrm{int}\} \\ \mathrm{invar}(c, e) &= l_1 \wedge l_2 \wedge \neg\,\mathrm{eq}(\mathrm{mgu}(l_2, \neg e))) \\ &= l_1 \wedge l_2 \wedge \neg(F = Field \wedge FN = Name \wedge T = \mathrm{int}) \end{aligned}$$

This invariant expresses what will be true after the removal of one or many fields about which we know statically only that their type is int and that they are from the same class as the fields addressed by the precondition c. □

5.3 Postconditions

The elementary effect expresses what will *additionally* be true after a transformation whereas the invariant expresses what will *still* be true, given that the precondition c was true before. Taken together, the effect and the invariant capture the *postcondition* of the transformation, that is, everything we know to be true after the transformation.

Definition 18 *(Elementary Postcondition). Let t be an elementary transformation and c be a condition that must be true immediately before t is executed. The* postcondition of c and t *is the disjunction of the invariant of c and t with the effect of t.*

$$\begin{aligned} \mathrm{post}_{ET} &: Cond \times ET \longrightarrow Cond \\ \mathrm{post}_{ET}(c, t) &= \mathrm{effect}(t) \vee \mathrm{invar}(c, \mathrm{effect}(t)) \end{aligned}$$

The disjunction expresses that each part, elementary effect and invariant, provides an independent contribution to the elementary postcondition. If the invariant is false, the effect might still hold and vice versa.

For transformation *sequences*, we must additionally take into account that later transformations are applied to the result of previous ones.

Definition 19 *(Cumulated Postcondition). Let t be a transformation sequence and c a condition that must be true immediately before t is executed. The* cumulated postcondition of t and c *is the postcondition of every transformation in the sequence applied to the postcondition of the preceding transformation.*

$$\begin{aligned} \mathrm{post} &: Cond \times T \longrightarrow Cond \\ \mathrm{post}(c, t) &= \begin{cases} \mathrm{post}(\mathrm{post}(c, t_1), t_2) & : t = t_2 \circ t_1 \\ \mathrm{post}_{ET}(c, t) & : t \in ET \end{cases} \end{aligned}$$

5.4 Cumulated Effect of a CT

The above general definition of cumulated postconditions is the basis for defining the cumulated effect and the cumulated postcondition of a CT.

Definition 20 *(Cumulated Effect and Postcondition of a CT). Let $CT \equiv c \rightarrow t$. The cumulated postcondition of CT is the cumulated postcondition of T and C. The cumulated effect of CT is the cumulated postcondition of T and true.*

$$\begin{aligned} \text{post}(CT) &= \text{post}(C, T) \\ \text{effect}(CT) &= \text{post}(\text{true}, T) \end{aligned}$$

The cumulated postcondition captures *everything* that we can derive about the state of a program after application of a CT. In contrast, the cumulated effect tells us the net effect of the CT's transformation on the program, ignoring what we can deduce from its precondition. This is important for interaction detection. There, we need to deduce how a CT's *transformation* influences other CTs.

Example 18 (Cumulated effect). Figure 4 illustrates all the CTs of our running example, depicting each CT as a stack of light boxes representing its precondition literals, above a stack of darker boxes representing its cumulated effect literals. Note that, for readability, "true" literals representing the individual effects of "newId" operations have been eliminated from the cumulated effects. Also, "concat(A, B, AB)" literals have been eliminated from the preconditions, replacing every occurrence of AB by the function term $A\&B$, denoting concatenation of A and B.

6 Interaction Detection

In this section, we introduce the notion of potential weaving interactions and show how their detection can be performed on the basis of CTs and CT effects. Our approach consists of a three-step process: creation of a graph of potential weaving interaction, its analysis for conflicts and insufficient information, and its topological sorting. If sorting is possible, it creates a weaving schedule that is guaranteed to preserve correctness and completeness of weaving (see Sect. 3.7).

6.1 Potential Weaving Interactions

According to the informal definition of weaving interactions in Sect. 3, the decision whether triggering or inhibition actually occurs can only be done with respect to a particular base program. This is too specific if we aim to verify whether a set of aspects is *always* interference free, no matter to which base programs it is applied. Aiming at a solution that is independent of any base program, we only check for *potential* triggering and inhibition.

Informally, $CT_1 \equiv c_1 \rightarrow t_1$ *potentially interacts* with $CT_2 \equiv c_2 \rightarrow t_2$ if execution of CT_1 on any program P *might* add, remove, or modify elements

such that c_2 additionally succeeds (or fails) for at least one previously false (or true) substitution.

For instance, the "uncountedFieldAccess" predicate (Example 12) in the precondition of `UseCnt` (Example 15) asks for field accesses and the existence of related counter fields. Thus, addition of a field access can potentially trigger this CT. However, if a suitable counter field is not added at the same time, we cannot know whether the CT is really triggered. The required counter field might or might not exist in the base program.

6.2 Deriving Potential Weaving Interactions from CTs

Detection of *potential* interactions requires one to determine whether a CT makes literals of another CT true (or false) by looking only at the CTs themselves but not at the programs they transform. This in turn requires the ability to determine which literals in the precondition of a CT change their truth values depending on whether another CT is executed before.[10].

Given a set of CTs $\{CT_1 \equiv c_1 \rightarrow t_1, ..., CT_n \equiv c_n \rightarrow t_n\}$ representing our weaving candidates, the construction of the weaving interaction graph starts by calculating the cumulated effect $e_i = \text{effect}(CT_i)$ for each $i = 1...n$, as explained in the previous section.

The positive literals in each cumulated effect e_i correspond to joinpoints potentially added by CT_i to a program. The negative literals correspond to potentially removed joinpoints. To derive potential triggering and inhibition, we compare each e_i with the preconditions of every candidate CT.

Definition 21 *(Potential triggering).* $CT_1 \equiv c_1 \rightarrow t_1$ potentially triggers $CT_2 \equiv c_2 \rightarrow t_2$ *iff* $\text{effect}(CT_1)$ *contains literals that are unifiable with literals from* c_2*:*

$$\text{triggers}_{\text{pot}}(CT_1, CT_2) \equiv \exists l_1, l_2 : l_1 \in \text{effect}(CT_1) \wedge l_2 \in c_2 \wedge \exists\, \text{mgu}(l_1, l_2)$$

Definition 22 *(Potential inhibition).* $CT_1 \equiv c_1 \rightarrow t_1$ potentially inhibits $CT_2 \equiv c_2 \rightarrow t_2$ *iff* $\text{effect}(CT_1)$ *contains literals whose negation is unifiable with literals from* c_2*:*

$$\text{inhibits}_{\text{pot}}(CT_1, CT_2) \equiv \exists l_1, l_2 : l_1 \in \text{effect}(t_1) \wedge l_2 \in c_2 \wedge \exists\, \text{mgu}(\neg l_1, l_2)$$

In the above two definitions, containment of a literal in a condition is to be understood recursively, that is, as if all literals for programmer-defined predicates had been replaced by their definitions.

Example 19 (Potential weaving interactions). Continuing Example 18, Fig. 4 illustrates the interactions detected by applying the above principles to our running example. The arcs represent the detected triggering (solid) and inhibition (dashed) interactions.

Note that a CT can potentially trigger and inhibit another one in multiple ways because of multiple pairs of unifiable literals.

[10] More precisely, we need to determine for which literals of the affected CT we get new substitutions (or lose substitutions)

7 Interference Prevention and Interference Diagnosis

In this section, we show how the information gained from the previous interaction analysis is used to diagnose weavability or different problem scenarios.

Diagnosis is performed on the basis of a weaving *interaction graph*. It contains a node for each candidate CT, labeled with the name of the CT, and an edge for each detected interaction, labeled with the action (addition or removal) and element that causes the interaction. Figure 5a shows a the weaving interaction graph created for the example from Fig. 4.

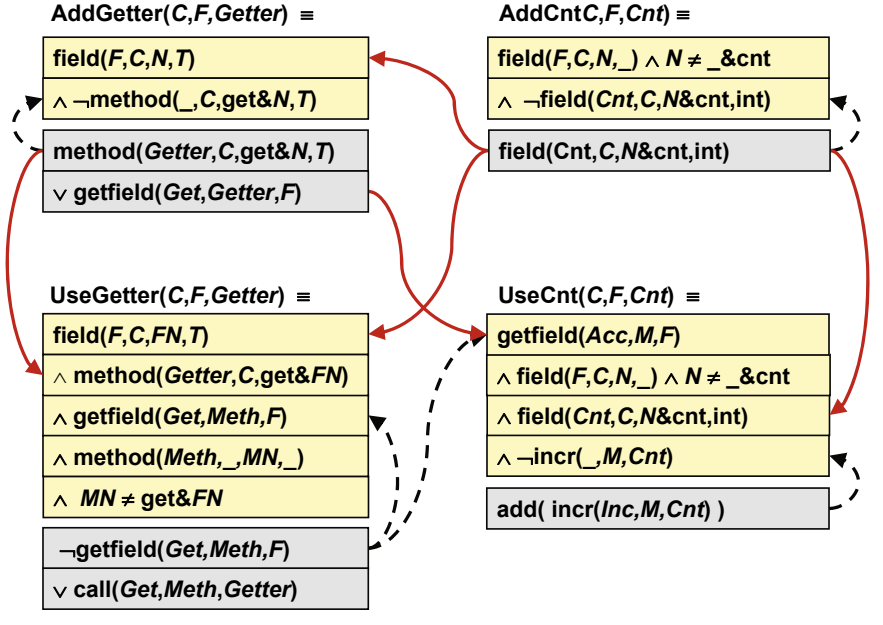
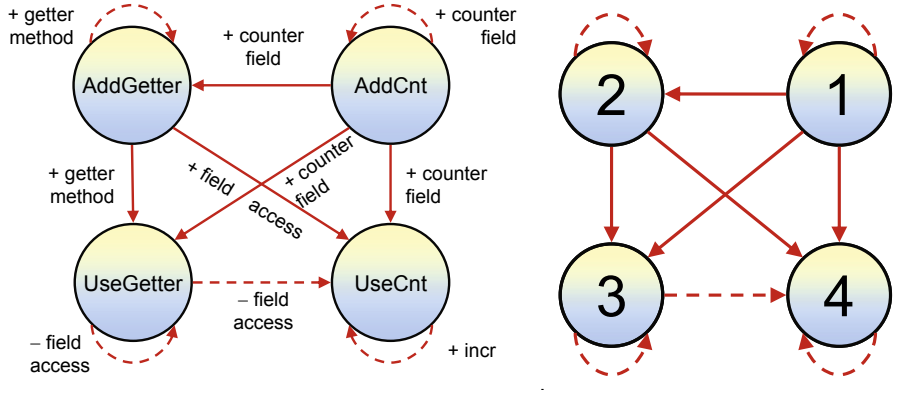


Fig. 4. Potential interactions for our running example: triggering (*solid arcs*) and inhibition (*dashed arcs*). *Light boxes* are condition literals, *gray* ones are effect literals. $A\&B$ denotes concatenation of A and B.

7.1 Interference Prevention

First of all, it is checked whether the interaction graph contains cycles. If not, the graph is sorted topologically. Figure 5b shows the total order computed for our example by topological sorting. Topological sorting yields an order that guarantees correctness and completeness of weaving (see Sect. 3.7). In particular, it guarantees that

- missed joinpoints (incomplete weaving) cannot occur because all CTs that potentially trigger others are executed before the triggered ones and
- erroneous joinpoints (incorrect weaving) cannot occur because all CTs that potentially inhibit others are executed before the inhibited ones.



a) Complete graph b) Topologically sorted

Fig. 5. Weaving interaction graph. (a) The symbol '+' indicates addition and the symbol '–' indicates deletion of an element. (b) The numbers 1–4 indicate the order determined by topological sorting, which guarantees complete and correct weaving.

7.2 Resolution of Semantic Interferences and Optimization

In our example, the weaving interaction graph defines a total order. However, in general, the graph can define a partial order and topological sorting can produce various total orders that respect the partial order. This enables use of additional criteria for determining a total order. In particular, the CTs that have no mutual order dependency can be ordered or parallelized based on

- criteria for resolving semantic interferences between them and
- criteria for optimization of weaving performance.

This provides a well-defined interface for integrating weaving interaction analysis with approaches for semantic interaction analysis and optimization.

7.3 Treatment of Cyclic Interactions

If the interaction graph contains cycles, we classify them according to the types of edges that they contain as inhibition cycles, triggering cycles, or mixed cycles:

Inhibition cycle. An inhibition cycle is a cycle consisting only of inhibition edges. A *self-inhibition* cycle is an inhibition cycle of length 1 (that is an inhibition edge whose source and target node are the same).

Triggering cycle. A triggering cycle is a cycle consisting only of triggering edges. If all arcs of a triggering cycle have the same kind of transformation literal as a source—in the expanded view of the dependency graph (Fig. 4)—we say that the cycle is monotonic. Otherwise, the triggering cycle is *mixed*. A monotonic cycle is *additive* (*destructive*) if the source literals of all arcs of the cycle are addition (deletion) operations.

Mixed cycle. A mixed cycle contains inhibition and triggering edges.

Each of these categories have different implications for the weaving process. They are explained below and summarized in Fig. 6. Figure 7 summarizes how the graph structure influences the required type of weaver.

Self-Inhibition. Self-inhibition indicates that the respective CT is written such that after it has been applied once to a certain joinpoint, it will not be applicable again to the same joinpoint. For instance, the self-inhibition of the `UseGetter` CT results from the fact that it adds a getter method but checks in its precondition that a getter method does not exist already. Since every CT should be written such that it is not infinitely applicable at the same joinpoint, self-inhibition is a highly desirable property of every node. If a node has no self-inhibition edge, we can issue a warning asking the programmer to add a suitable check to the CTs precondition.

Long Inhibition Cycles. Inhibition cycles of length greater than one indicate a *conflict*. No matter with which CT in the cycle we start we will get a different weaving result. Long inhibition cycles result from an improper joint use of multiple CTs. They are plain errors that are reported to the programmer asking to change the definition of some CTs or their composition.

However, change of individual CTs/aspects might not be possible, for instance, because they are from a third party. Change of the composition might not be possible because no functionally similar CTs are available that could be used instead of the ones that cause problems. If the conflict cannot be resolved, the program is not weavable.

Mixed Cycles and Mixed Triggering Cycles. This category corresponds to cases where we cannot make any prediction. Iterative weaving with a weaver that is able to react to the type of events in the cycle can or cannot terminate, and

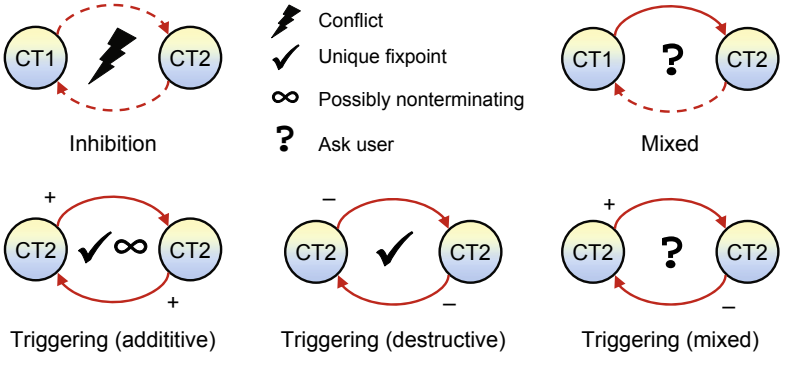


Fig. 6. Types of cycles and their implications on weavability. The '+' and '–' symbols attached to the arcs indicate the type or operations (additions or deletions) that produce the respective dependency.

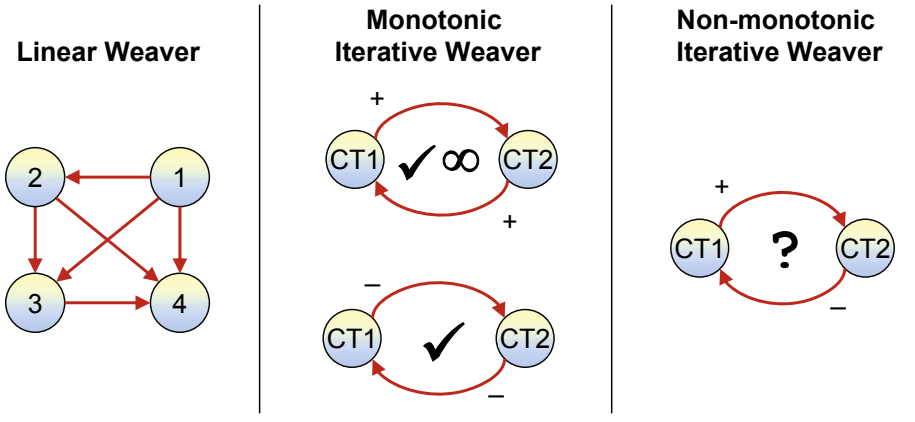


Fig. 7. Implication of graph structure on required type of weaver (the symbols are the same as in the previous figure

if it terminates, it can or cannot lead to a unique fixpoint that is independent of where in the cycle we started.

In contrast to the previous category which indicated plain errors, these cases indicate *incomplete programs* in the sense that there is not enough information available to decide automatically. This is where research on language features that provide additional information to the analysis will be useful.

Monotonic Triggering Cycles. Provided that the weaving process terminates, an additive triggering cycle is guaranteed to produce a unique fixpoint, no matter where in the cycle we started. A destructive triggering cycle additionally guarantees termination. Both cycles can be woven using a monotonic iterative weaver that is able to react to the respective events.

These cases are neither errors nor do they require additional specifications in the program. However, they require a more powerful weaving infrastructure.

Cycle Summary. The categorization introduced above enables precise definition of interferences and conflicts in contrast to "normal" interactions.

Definition 23 *(Interference, Conflict, Incomplete Program, Weavability).* *An* interference *is any cycle in the interaction graph. A* conflict *is an inhibition cycle of length greater than 1. A set of candidate CTs is* incomplete *if its interaction graph contains a mixed cycle or a mixed triggering cycle. It is* unweavable *if it is incomplete or the graph contains a conflict.*

7.4 Creation of a Weaving Schedule

The analysis of the interaction graph stops if it detects that the candidate set is unweavable. Because the interaction graph contains information about the action

and element that caused each of its arcs, it is possible to issue error messages that contain precise diagnostics about the source of a problem.

For weavable graphs, the algorithms proceeds to the creation of a weaving schedule. A *weaving schedule* specifies the execution of a candidate CT set. It can specify that certain candidates may be executed in parallel and that others must be executed in a specific order. In addition, it can specify that certain parts of the schedule must be iterated.

The weaving schedule is constructed in three steps. First, monotonic triggering cycles are collapsed into a single metanode. The metanode is annotated with the type of the cycle, which determines the type of the required weaving algorithm. Then, self-inhibition cycles are deleted from from the graph since they have no meaning at weave time. Finally, the resulting acyclic graph that possibly encapsulates monotonic cycles in its metanodes is sorted topologically. This step can exploit additional criteria for scheduling CTs with no mutual order dependencies (see Sect. 7.2).

If the graph contains no metanodes, topological sorting yields an order that guarantees correctness and completeness of weaving even when executed with a simple, linear weaver. If the graph contains metanodes, the weaving schedule interleaves the overall linear weaving with iterative weaving phases for the metanodes.

Note that in contrast to the ad-hoc interference repair via complex cascading weavers (Sect. 3.7), our analysis enables weaving of monotonic triggering cycles on a much simpler infrastructure. *Iterative weavers* only need the ability to repeatedly execute a portion of the schedule. They do not need the expensive monitoring infrastructure for detecting interferences at run time, since they work based on the statically created schedule. Furthermore, they do not need the ability to undo preceding actions, since, for monotonic cycles, iteration is guaranteed to achieve a fix point. Last, but not least, they are much more efficient since any action that they perform in addition to what a linear weaver would do are confined to the subgraph that really needs them.

8 Inlining Versus Forwarding

In this section, we discuss how our analysis applies to the two semantically equivalent implementation techniques that can be used by a weaver: inlining and forwarding (Fig. 8). *Inlining* is the weaving technique analyzed so far. It performs each aspect action precisely a the related joinpoint. For instance, line 1 of Fig. 8c shows how the body of the advice from Fig. 8b is inlined before the joinpoint in line 2.

In contrast, *forwarding* is based on weaver-generated methods that encapsulate the advice body *and* the joinpoint to which the advice is applied. All occurrences of the original joinpoint are replaced by invocations of the respective forwarding method. Figure 8d shows in line 1 and 3 the invocations of the forwarding method defined in lines 4–6. Forwarding is used extensively in AspectJ.

a) Excerpt of base program:

```
1 temp = f;
2 ...
3 if (m().f == null) ...
```

b) Advice:

```
1 before(Base t) : get(int *.f) {
2   while (...) ...;
3 }
```

c) Inlined weaving:

```
1 while (...) ...;       // inlining
2 temp = f;
3 ...
4 if (m().f == null) ... // inapplicable
```

d) Forwarding-based weaving:

```
1 temp = $$$_forw_f();
2 ...
3 if (m().$$$_forw_f() == null) ...

4 int $$$_forw_f() {
5    while (...) ...; // advice body
6    return f;        // join point f
7 }
```

Fig. 8. Weaving by inlining or forwarding

Note how both weaving techniques implement the same semantics. They differ, however, in their applicability, run time efficiency, and their influence on the occurrence of weaving interferences.

The main argument for inlining is efficiency. Pawlak [48] compares the runtime of AspectJ forwarding methods and inlined advice for a normal method call, for code accessing the static joinpoint context, and for code accessing the dynamic joinpoint context. He demonstrates that inlined weaving needs 83 ms in all three cases whereas AspectJ's forwarding needs 193, 787, and 8570 ms, respectively.

The main argument for forwarding is generality. It is also applicable in cases where joinpoints occur in expressions, e.g., in the condition of an if statement, as in line 3 of Fig. 8d. Inlining is not applicable in such a case (see line 4 of Fig. 8c) because in most object-oriented languages (e.g., Java, Eiffel, C++) it is not possible to inline a statement into an expression. Note that here it is not possible to inline the while statement *before* the if statement because that would also be before the invocation of the method m in line 4.

Joinpoint Addition Versus Erasure. Regarding weaving interactions, each block of inlined code can add further joinpoints to the program. The implications of such added joinpoints have been analyzed in the previous sections.

Forwarding does not add joinpoints because the weaver-generated forwarding method names are chosen so that they cannot be matched by pointcuts, as indicated in Fig. 8d by the $$$_ prefix of the forwarding method. Thus, forwarding effectively erases all matching joinpoint occurrences in the base program and in other forwarding methods, replacing them by a single joinpoint occurrence in the body of the forwarding method (Line 6 of Fig. 8d. Therefore, we say that forwarding has the effect of *joinpoint erasure*.

Joinpoint erasure has an interesting implication on weaving interactions. Because it prevents addition of new joinpoints, it prevents occurrence of interferences based on triggering cycles. Indeed, missed triggering does not occur in AspectJ, which uses forwarding extensively. Unfortunately, forwarding does not

a) An aspect:

```
1 aspect BeforeF1 {
2   before(Base t) : get(int *.f1) {
3     f2;
4   }
5 }
```

b) Another aspect:

```
1 aspect BeforeF2 {
2   before(Base t) : get(int *.f2) {
3     f1;
4   }
5 }
```

c) BeforeF1 woven with forwarding:

```
1 int $$$_forw_f1() {
2   $$$_forw_f2();
3   return f1;
4 }
```

d) BeforeF2 woven with forwarding:

```
1 int $$$_forw_f2() {
2   $$$_forw_f1();
3   return f2;
4 }
```

Fig. 9. StackOverflowExceptions can be caused by forwarding cycles spanning an arbitrary number of advices and aspects

really solve the problem of missed triggering, it just promotes weaving problems to run time problems. This is easy to see if we assume that the advice body in line 2 of Fig. 8b contains itself an access to f. Then, line 5 of Fig. 8d would contain a recursive invocation of $$$_forw_f. Thus, after an apparently successful weaving, the woven program's execution would lead to an infinite loop.

Indeed, this problem is known and documented in the "5. Pitfalls" section of the AspectJ programming guide, claiming that "*the overall problem is advice applying within its own body. There's a simple idiom to use if you ever have a worry that your advice might apply in this way. Just restrict the advice from occurring in joinpoints caused within the aspect.*" Unfortunately, this explanation attempt is wrong. The problem is *not* restricted to advice applying within its *own* body. Figure 9 shows that it can be caused by forwarding cycles spanning two advices in two aspects which is straightforward to generalize to an arbitrary number of advices and aspects. For uniformly generic aspects that do not encode hardwired assumptions about the base program [27], scenarios as the one in Fig. 9a and b are common.

For this general problem, the suggestion to "*restrict the advice from occurring in joinpoints caused within the aspect*" does not help. To prevent the loop manually by extending the pointcut specification, one would need to know in advance all the other aspects contributing to the cycle. This typically is not the case in the open-world scenario that was the motivation for our work (see Sect. 1.1). Even if we operated in a closed world, a manual extension of pointcuts would be a very bad idea since it would hardwire previously implicit dependencies into the aspect code and would make the pointcuts unreadable, hiding their true meaning behind much noise.

Conclusion. We can conclude that forwarding trades a *potentially* nonterminating weaving process caused by cyclic triggering for a *definitely* nonterminating invocation loop at run time caused by mutually recursive method invocations. Therefore, our analysis is equally essential for systems that perform forwarding as it is for systems based on inlining.

Our technique guarantees completeness and correctness of weaving (Theorem 1), not just for forwarding based but *also* for inlined weaving. Thus, it opens the door for weavers that produce interference-free, yet highly efficient, inlined aspect code. For forwarding based weavers, our analysis helps to statically diagnose and reject forwarding cycles that will lead to infinite recursion at run time. Detection of inhibitions is essential for forwarding based weaving too because forwarding does not prevent erroneous joinpoints, as demonstrated by the experiment with AspectJ reported in Sect. 2.2.

9 Discussion and Extensions

In this section, we discuss

- the concept of interleaved aspect weaving enabled by our approach,
- its implications on language design,
- the relation of our work to the often-discussed "shared joinpoint" problem, and
- the complexity and experimental run time performance of our approach.

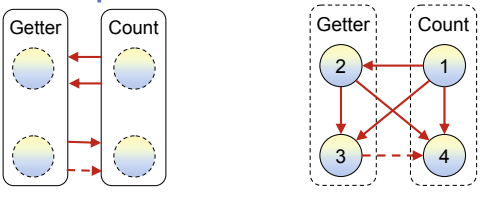
9.1 Interleaved Aspect Weaving

An analysis at the level of *aspects* would have failed in our example. As shown in Fig. 10a, aspect-level analysis yields an unweavable cyclic dependency. If we insist on executing all CTs of one aspect together, the example cannot be woven correctly and completely.

The only solution in this case is *interleaved* weaving of the aspects, illustrated in Fig. 10b. The weaving order computed by our approach starts with a CT of the counter (`AddCounter`), continues with the two CTs of the getter, and, finally, executes the second part of the counter (`UseCounter`).

9.2 Language Design Issues

The need for interleaved aspect weaving shows that analysis at aspect level is too coarse grained. The same is true for any language elements that define ordering



a) Granularity = Aspect b) Granularity = CT

Fig. 10. Analysis at aspect level yields unweavable cyclic dependencies. Analysis at CT level resolves the cycle, producing an order that interleaves CTs from different aspects.

of *aspects*. Instead, we need explicit language elements that define ordering of *AESs*, that is, of CTs.

To limit the assumptions that aspects make about other aspects, the ordering statements must only define *relative* ordering of two AESs. The sum of such relative declarations defines a partial order that can be extended to total order by the weaver, as we have shown. Relative ordering also avoids to needlessly constrain the joint deployment of multiple aspects.

Programmer-specified ordering statements can provide missing information that we cannot infer from the minimalist assumptions that we made, be redundant to our inferred ordering, or be inconsistent with it. In the case of inconsistencies, the analysis can issue warnings that the aspect's implementation does not comply with its stated intention.

9.3 Shared Joinpoints

Most work on aspect interactions and interferences has focuses on interactions of advice at shared joinpoints. This is too narrow since it fails to identify the real cause of many errors.

Some papers even claim that interactions and interferences can *only* happen at shared joinpoints [13, 49]. This is wrong. An AESs that triggers or inhibits another by *adding* elements never acts at the same joinpoint as the affected one (see the examples analyzed in this paper). At joinpoint a it adds an element b that might be a joinpoint or contribute to the selection of a joinpoint c by another aspect.

The situation is more subtle if an aspect *removes* joinpoints. Then it always selects the joinpoint that it removes. An aspect that is affected by the removal necessarily selects the same joinpoint. Since in this case both aspects select the same joinpoint it appears that this is an instance of the often-cited "shared joinpoint" problem. However, we think it is not. Most papers that talk about interferences at shared joinpoints mean interferences at a semantic level, arising from the fact that multiple aspects or the aspects and the base share and jointly modify part of the program's *state*. This is very different from modifying the program's *structure*. Correctness and completeness of weaving ensures proper structure. This in turn is a necessary but not sufficient condition for proper behavior.

These two levels need to be clearly separated to properly understand the mechanics of interferences. We have that

- weaving interferences can arise between weaving candidates that select the same *or different* joinpoints,
- interferences at shared joinpoints can have a structural *and* a semantic facet.

9.4 Detection of Violated Ordering at Shared Joinpoints

The structural facet of the shared joinpoint problem is a special case of weaving interference. It arises from the fact that aspects make assumptions not just

about the existence of joinpoints in a program but also about their order. In this section, we show that a slight extension of our condition language suffices to detect ordering conflicts at shared joinpoints with the mechanisms introduced in the previous section. If ordering information for the statements of a program is available it turns out that interference at shared joinpoints actually burns down to a long inhibition cycle caused by conflicting ordering assumptions.

We represent explicit ordering information by a binary predicate "before" that takes program element identities as arguments. A fact before($First$, $Second$) states that the element with identity $First$ comes *immediately* before the element with identity $Second$. Note that if the semantics would be "somewhere before" instead of "immediately before" then any order of multiple advices for the same joinpoint would be acceptable. A conflict between different advices that specify an action before the same joinpoint can only be derived if each advice assumes that its effects come immediately before the joinpoint. Note further, that from a semantic point of view, interferences could arise even between statements that are not immediately before. Again, it is important to always be aware of the level (structural or semantic) of the discussion.

A before advice is translated to an order-sensitive CT by using the before predicate in two ways[11]:

- To avoid infinite application of the advice, the condition must state that the effect of the advice must not already be present before that joinpoint. This would typically involve a negated literal, ¬ before($Stment$, $JoinPoint$).
- The transformation part must state the precise location of the effect relative to the joint point. This is done by adding a before($Stment$, $JoinPoint$) fact.

Figure 11 shows an order-sensitive definition of the CT `UseCnt`. Compared to the definition from Sect. 4.6, its precondition is refined to state that the CT is only applied if there is not already an increment before the field access, and the transformation is extended to include the information that the increment of the counter is added *immediately before* the access to the field F.

Translation of any other order-sensitive aspect language elements to CTs will have the same structure: the CTs always refer to a joinpoint using negated before($Statement$, $JoinPoint$) literals in the precondition and positive ones in the transformation.[12] Thus, our analysis will detect a mutual inhibition (conflict) of such CTs.

Note that different "before" literals that contain *only* variables will always unify with each other. Thus, an analysis that is independent of any base program would just detect that *all* order-sensitive actions might potentially interfere with all other, which is not really helpful information. Therefore, an analysis of interactions between CTs that contain "before" literals should preferably be performed with reference to a specific base program, that is, after computing substitutions for the variables of each CT by evaluating the CT's preconditions on the base program.

[11] Translation of after advice is possible using the "before" predicate with reversed argument order, before($JoinPoint$, $Statement$).

[12] Or vice-versa, if the aspect can *delete* program elements.

$$
\begin{array}{ll}
\text{UseCnt}(F, C, Cnt) \equiv & \\
\quad \text{exists}(\text{getfield}(Acc, M, F)) & \text{// Method } M \text{ contains an access} \\
\quad \wedge\, \text{exists}(\text{field}(F, C, N, \_)) & \text{// ... to the field } F \text{ of class } C \\
\quad \wedge\, \neg\, \text{concat}(\_, \_\text{cnt}, N) & \text{// } F \text{ is no counter} \\
\quad \wedge\, \text{concat}(N, \_\text{cnt}, CntN) & \text{// } CntN \text{ is the name of } F\text{'s counter} \\
\quad \wedge\, \text{exists}(\text{field}(Cnt, C, CntN, \text{int})) & \text{// } Cnt \text{ is } F\text{'s counter defined in } C \\
\quad \wedge\, \neg \Big(\text{exists}(\text{incr}(Inc, M, Cnt)) & \text{// } Cnt \text{ is not incremented in } M \\
\qquad \underline{\wedge\, \text{before}(Inc, Acc))} \Big) & \text{// ... before the field access } Acc \\
\quad \longrightarrow & \\
\quad \circ\, \text{add}(\text{incr}(Inc, Meth, Cnt)) & \text{// add an increment of } Cnt \\
\quad \underline{\circ\, \text{add}(\text{before}(Inc, Acc))} & \text{// ... immediately before } Acc \\
\quad \text{newId}(Inc) &
\end{array}
$$

Fig. 11. A version of the CT `UseCnt` from Sect. 4.6 that represents statement order information via the "before" predicate

9.5 Run Time Efficiency

Our analysis compares the precondition of every CT and the effect of every CT. If the total number of CTs to which an aspect program translates is N, then our algorithm performs N^2 comparisons. Prior to each comparison, we must calculate the effect of a CT. This is linear in the number of literals in the transformation of the CT. Each comparison means attempting to unify every literal of the transformation effect with some literal of another CT's precondition. Assuming that the maximum number of literals in any precondition or transformation is M, the number of attempted unifications is clearly bounded by M^2. Unification is linear in the number of variables of the unified literals.

Thus, if we consider the assignment of a variable to another as the smallest step in our algorithm and assume that V is the maximum number of variables in any literal, we can state that the number of steps performed for analyzing an aspect program is in the order of $O(N^2) * O(M^2) * O(V)$ in the worst case. Since M and V are typically low, the complexity is dominated by the $O(N^2)$ part. Thus, we can conclude that, in the average case, our algorithm is quadratic in the number of advices and introductions in an aspect program. These results are very good, given that many other analyzes have inherently exponential complexity (see section on related work). Note also that the complexity of our algorithm is independent of the size of the base program.

The good performance expected from the complexity analysis is confirmed by run time measurements of the implementation of our approach in the Condor tool [50]. In cases where we analyzed "real" CTs, the analyzes were carried out in less than a second, including the computation of all interference-free orders

(if possible) or the detection and categorization of all cycles [51]. To verify the scalability of our algorithm, we randomly generated nontrivial CTs and run the analysis on several hundreds of them. The total run time remained below 1 min although the implementation of our algorithm in Condor is rather straightforward and still provides many opportunities for optimization.

10 Related and Future Work

In this section, we discuss the relation of our approach to other work in this domain and related open issues.

Aspect and feature interactions and interferences have been described by many authors [2, 4, 6–10, 12–20, 24–26, 52–55]. Still, we are not aware of any previous discussion of the completeness and correctness of aspect weaving and of the related weaving interactions and interferences.

In many related approaches, conflicts are at a semantic level and are considered to be unordered executions of advice at shared joinpoints. We refute the claim that non-shared joinpoints are *sufficient* to ensure that aspects do not interact, address the overlooked issue of weaving interferences, and show that part of the shared joinpoint problem is a special case of weaving interferences (see Sect. 9.3).

Kienzle et al. [14] also investigate dependencies and weavability. However, their dependencies relate to the use of names from other modules, not to weaving dependencies. Their notion of weavability is defined at aspect level and excludes cyclic dependencies whereas ours is able to detect that aspect-level cycles can be resolved at the finer-grained level of aspect effects. Moreover, we allow weaving of certain kinds of cyclic dependencies at the fine-grained level.

Havinga et al. [56] address dependencies that can be caused by using pointcut languages combined with the introduction of elements (e.g., annotations) in the structure that is queried by the pointcut language. Although the issue of weavability is not addressed explicitly, this approach shares some of our ideas. Our analysis is more general since it is equally applicable to introductions and advice. To identify an order that resolves potential ambiguities [56], apply their analysis to all possible orders of introductions. This results in $O(N!)$ complexity, if N is the number of introductions in a program. Our approach is quadratic in the number of CTs to which a set of aspects translate (see Sect. 9.5).

In [57], Havinga et al. propose a refined analysis that uses graph transformations and the GROOVE (Graphs for Object-Oriented Verification) tool set [58, 59]. Although the approach is still very different from ours, the graph based formalization makes a lot of remarkable parallels explicit. Investigating the correspondence of logic and graphs in this particular context will be a rewarding topic for future work.

Significant advances have been achieved regarding the detection and resolution of semantic interferences [6, 7, 12, 13, 16, 17, 22, 24, 52, 60–63]. Many of these approaches use heavyweight techniques (model checking, state machines, logics of time and events, formal verification of constraints, or exploration of

the entire space of possible aspect orderings). Some require additional behavioral specifications or language extensions. Our approach is rather lightweight, requiring none of the above. A worthwhile topic of future work will be the exploration of how our technique can be gradually combined with more heavyweight ones to cover also semantic interactions.

Another domain that requires additional exploration is the impact of weaving interactions to dynamic weaving. Because the added power of dynamic weaving is the ability to weave based on dynamic state and event history, this will be closely related to the previously mentioned investigation of the relation to semantic interferences.

Dürr et al. [60, 61, 64] present an approach to the analysis of interferences among composition filters that requires no additional specifications in the aspect program. The relevant information is automatically derived from specifications of the effects of the different composition filter types. However, when applied to other languages, the approach requires explicit inclusion of additional specification (e.g., in the form of annotations).

Our analysis differs from many others in that it is independent of the base program, making positive analysis result generally applicable.[13] In addition, it makes the analysis independent of the size of the base program, thus enabling faster analyzes than in the case where program-specific techniques are used (e.g., program slicing as used by Blair and Monga [9]). We are aware of only three other approaches that perform interference analysis independently of any base program (reviewed next).

Douence et al. [13, 49] present a logic-based approach that works equally well in a base program-independent and base program specific mode. They provide an expressive dynamic crosscut language and propose linguistic support for conflict resolution. Their approach is extended to stateful aspects in [7]. The discussion in [13, 49] includes a "silent" weaver that makes aspect effects invisible to other aspects. We discuss the case when aspect effects *are* visible to other aspects. Our discussion shows that hiding aspect effects from other aspects does not prevent but *provokes* mutual constraint violations. For instance, if we hide the counter's effects from the accessor, counting will always undermine thread safety, since the assumption that all accesses, including those to the counter, are via synchronized accessor methods will necessarily be violated (Sects. 2 and 3).

Krishnamurthi et al. [24] propose an analysis of aspect-base interactions using model-checking techniques. A set of properties that should be preserved in a base program is the pointcut descriptor of an advice, the advice is checked in isolation, without needing access to the base program. The analysis does not need to be repeated if the implementation of the advice changes.

A somehow similar approach is presented by Goldman and Katz [65]. Their model-checking approach is able to generically verify that an aspect will not interfere with any program that fulfills a set of assumptions. Aspects and assumptions expressed in LTL are represented as state machines. Like the approach of [65] and ours, the method of Goldman and Katz lets the base program

[13] Aspects that have no *potential* interferences will not interfere on *any* base program.

be *oblivious* [1] of aspects. Unlike our approach, it analyzes aspect-base interferences and restricts the aspects to be *weakly invasive* [66].

Our weaving interferences correspond in a more general setting to interferences between program transformations. We are aware of only one other tool with the ability to perform transformation dependency analysis in a language-independent manner: *AGG* [67], a system based on graph transformations. In [51], *AGG* is compared to *Condor*, our implementation of the principles explained in this paper. The comparison showed that CTs are more expressive than graph several orders of magnitude faster. In AGG, it took more than 15 minutes to compute results for which Condor required less than a second. It will be interesting to transfer the analysis functionality of AGG to the GROOVE tool set [58, 59] and repeat the comparison. GROOVE uses algorithms that can do graph matching and isomorphism detection in polynomial time in most cases.

With the exception of the JMangler system [3, 68], it seems that there has been no prior discussion of the option to use iterative weaving mechanisms for interference repair. Discontinuous weaving is proposed for morphing aspects [69]. However, the incremental weaving steps used there are motivated by the desire to gain efficiency via lazy dynamic weaving. They do not target the repair of weaving interferences.

11 Conclusions

In this paper, we have addressed the problem of interferences between independently developed but jointly deployed, black-box aspects. We have defined the notion of complete and correct aspect weaving and have demonstrated that weaving interactions (triggering and inhibition) can give rise to interferences (missed joinpoints and erroneous effects) that compromise the correctness and completeness of weaving. We have shown that weaving interferences can lead to hard-to-trace semantic misbehavior, violations of nonfunctional requirements, and incorrect program structure.

As a first step toward a solution, we have introduced a representation of aspect effects (advice and introductions) as conditional program transformations (CTs). CTs are a logic foundation for program transformations that are guarded by preconditions. Based on this representation, we have developed an efficient, automated solution to weaving interaction detection, weaving interaction resolution, and weaving interference detection.

In addition, we have pointed out that the prevention of interferences by computing a suitable weaving order in advance is not the only possible option. We have introduced different weaving strategies (linear, monotonic, nonmonotonic) for interference repair at run time. Our analysis generates a precise weaving plan and identifies the least complex class of weaver that suffices for weaving a particular subset of a given set of aspects.

Our technique is able to detect and automatically resolve problems that are not addressed by any other aspect interference analysis approach that we know of. This has been demonstrated on the example of two mutually interdependent aspects for which our technique is able to compute an interference-free weaving plan. In this context, we have introduced the idea of *interleaved* weaving of aspects and have shown that it enables weaving in scenarios that have been previously considered to be unweavable. If necessary, for interaction resolution, our analyzes compute interleaved weaving plans.

Last, but not least, we have shown that the complexity of the proposed algorithm is quadratic in the number of analyzed conditional transformations and have reported the result of concrete run time measurements. Beyond being itself efficient, our analysis technique enables use of an efficient linear weaver whenever an interference-free order can be computed. In addition, it is the basis for weavers that generate high-performance code by extensive inlining of aspect effects. For weavers that use forwarding instead of inlining, we have shown how to detect statically infinite forwarding cycles. The analysis is independent of the base programs to which the analyzed aspects are applied, making the results reusable in arbitrary contexts.

Acknowledgements

The author wants to thank the anonymous reviewers whose knowledgeable and often-challenging comments have helped to improve the presentation of this paper in many ways. Great thanks are due to Shmuel Katz for his guidance in the editing process. Andrew Clement has reacted quickly to my questions regarding the misbehavior of AspectJ described in Sect. 2.2.

References

[1] Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns (OOPSLA 2000) (2000)

[2] McEachen, N., Alexander, R.: Distributing classes with woven concerns – an exploration of potential fault scenarios. In: Tarr, P. (ed.) AOSD 2005 – 4th International Conference on Aspect-Oriented Software Development, Chicago, USA. ACM Press, New York (2005)

[3] Kniesel, G., Costanza, P., Austermann, M.: JMangler - a framework for load-time transformation of Java class files. In: First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), pp. 100–110. IEEE Computer Society Press, Los Alamitos (2001)

[4] Ostermann, K., Kniesel, G.: Independent extensibility—An open challange for AspectJ and Hyper/J. In: Workshop on Aspects and Dimensions of Concerns (ECOOP 2000) (2000)

[5] Szyperski, C.: Independently extensible systems – software engineering potential and challenges. In: Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, Computer Science Association (1996)

[6] Bergmans, L.M.: Towards detection of semantic conflicts between crosscutting concerns. In: [70], http://www.st.informatik.tu-darmstadt.de:8080/ecoop/workshops/16.phtml
[7] Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: Lieberherr, K. (ed.) AOSD 2004 – 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK, University of Lancaster, pp. 141–150. ACM Press, New York (2004)
[8] Tessier, F., Badri, M., Badri, L.: A model-based detection of conflicts between crosscutting concerns: Towards a formal approach. In: Huang, M., Mei, H., Zhao, J. (eds.) International Workshop on Aspect-Oriented Software Development (WAOSD 2004), Beijing University, China (2004)
[9] Blair, L., Monga, M.: Reasoning on AspectJ programmes. In: [71], pp. 45–49
[10] Prehofer, C.: Feature interactions in statechart diagrams or graphical composition of components. In: Kandé, M., Aldawud, O., Booch, G., Harrison, B. (eds.) Second International Workshop on Aspect-Oriented Modeling with UML (<<UML>>2002) (2002)
[11] Pang, J., Blair, L.: An adaptive run time manager for the dynamic integration and interaction resolution of features. In: Akşit, M., Choukair, Z. (eds.) Proc. 2nd Int'l Workshop on Aspect Oriented Programming for Distributed Computing Systems (ICDCS 2002), vol. 2 (2002)
[12] Clifton, C., Leavens, G.T.: Observers and assistants: A proposal for modular aspect-oriented reasoning. In: [72], pp. 33–44
[13] Douence, R., Fradet, P., Südholt, M.: Detection and resolution of aspect interactions. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 173–188. Springer, Heidelberg (2002)
[14] Kienzle, J., Yu, Y., Xiong, J.: On composition and reuse of aspects. In: [73]
[15] Skotiniotis, T., Lieberherr, K., Lorenz, D.: Aspect instances and their interactions. In: [73]
[16] Sihman, M., Katz, S.: Model checking applications of aspects and superimpositions. In: [74]
[17] Sihman, M., Katz, S.: A calculus of superimpositions for distributed systems. In: Kiczales, G. (ed.) Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD 2002), pp. 28–40. ACM Press, New York (2002)
[18] Störzer, M., Krinke, J.: Interference analysis for AspectJ. In: [74]
[19] Störzer, M.: Analysis of AspectJ programs. In: [71], pp. 39–44
[20] Störzer, M., Krinke, J., Breu, S.: Trace analysis for aspect application. In: [70], pp. 39–44, http://www.st.informatik.tu-darmstadt.de:8080/ecoop/workshops/16.phtml
[21] Alexander, R.T., Bieman, J.M.: Challenges of aspect-oriented technology. In: Workshop on Software Quality, 24th Int'l Conf. Software Engineering (2002)
[22] Clifton, C., Leavens, G.T.: Obliviousness, modular reasoning, and the behavioral subtyping analogy. In: [73]
[23] Kienzle, J., Yu, Y., Xiong, J.: On composition and reuse of aspects. In: [74]
[24] Krishnamurthi, S., Fisler, K., Greenberg, M.: Verifying aspect advice modularly. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering (2004)
[25] Zhao, J.: Towards formal verification of aspectj programs. In: Proc. 14th Conference of Japan Society for Software Science and Technology (JSSST 2003), Aichi, Japan (2004)

[26] Zhao, J., Rinard, M.: Pipa: A behavioral interface specification language for AspectJ. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 150–165. Springer, Heidelberg (2003)
[27] Günter Kniesel, T.R.: A definition, overview and taxonomy of generic aspect languages. L'Objet, Special issue ' Développement de logiciels par aspects 12(2-3) (2006)
[28] Kniesel, G., Koch, H.: Static composition of refactorings. Science of Computer Programming (Special issue on Program Transformation) 52(1-3), 9–51 (2004), http://dx.doi.org/10.1016/j.scico.2004.03.002
[29] Kniesel, G.: A Logic Foundation for Conditional Program Transformations. Technical report IAI-TR-2006-01, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany (2006)
[30] JTransformer homepage (2006), http://roots.iai.uni-bonn.de/research/jtransformer/
[31] CTC project homepage (2006), http://roots.iai.uni-bonn.de/research/ctc/
[32] Günter Kniesel, T.R.: Generic aspect languages - needs, options and challenges. In: Seinturier, L. (ed.) Journée Francophone sur le Développement de Logiciels par Aspects (2005)
[33] Rho, T., Kniesel, G.: Uniform Genericity for Aspect Languages. Technical report IAI-TR-2004-4, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany (2004)
[34] Kniesel, G., Rho, T., Hanenberg, S.: Reusable Pattern Implementations Need Generic Aspects (2004), http://www.disi.unige.it/person/CazzolaW/RAM-SE04.html
[35] Kniesel, G., Bardey, U.: An analysis of the correctness and completeness of aspect weaving. In: Proceedings of Working Conference on Reverse Engineering 2006 (WCRE 2006). IEEE, Los Alamitos (2006)
[36] Hanenberg, S., Unland, R.: Parametric introductions. In: Akşit, M. (ed.) Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD 2003), pp. 80–89. ACM Press, New York (2003)
[37] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: Getting started with AspectJ. Comm. ACM 44(10), 59–65 (2001)
[38] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
[39] Bergmans, L., Aksit, M.: Principles and design rationale of composition filters. In: Filman, R., Elrad, T., Clarke, S., Aksit, M. (eds.) Aspect-Oriented Software Development. Addison-Wesley, Reading (2004)
[40] Bergmans, L., Akşit, M.: Composing crosscutting concerns using composition filters. Comm. ACM 44(10), 51–57 (2001)
[41] Aksit, M., Bergmans, L., Vural, S.: An object-oriented language-database integration model: The composition-filters approach. In: Lehrmann Madsen, O. (ed.) ECOOP 1992. LNCS, vol. 615, pp. 372–395. Springer, Heidelberg (1992)
[42] Ossher, H., Tarr, P.: The shape of things to come: Using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. Comm. ACM 44(10), 43–50 (2001)
[43] Ossher, H., Tarr, P.: Multi-dimensional separation of concerns using hyperspaces. Technical Report 21452, IBM Research Report (1999)
[44] Rajan, H., Sullivan, K.J.: Classpects: unifying aspect- and object-oriented language design. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 59–68. Springer, Heidelberg (2006)

[45] Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join-points in aspect-oriented programming. In: [72], pp. 1–8
[46] Windeln, T.: LogicAJ - Eine Erweiterung von AspectJ um logische Meta-Programmierung. Diploma thesis, CS Dept. III, University of Bonn, Germany (2003)
[47] Bardey, U.: Abhängigkeitsanalyze für Programm-Transformationen. Diploma thesis, CS Dept. III, University of Bonn, Germany (2003)
[48] Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program analysis and transformation in java. Research Report 5901, INRIA (2006)
[49] Douence, R., Fradet, P., Südholt, M.: Detection and resolution of aspect interactions. Technical report RR-4435, INRIA (2002)
[50] Condor homepage (2006), http://roots.iai.uni-bonn.de/research/condor/
[51] Mens, T., Kniesel, G., Runge, O.: Transformation dependency analysis - a comparison of two approaches. L'Objet, Special issue 'Langages et Modèls à Objets 2006 12(HS), 167–182 (2006)
[52] Katz, S.: A survey of verification and static analysis for aspects. Technical Report AOSD-Europe Milestone M8.1, AOSD-Europe-Technion-1, Technion Israel (2005), http://www.aosd-europe.net/deliverables/m8-1ver.pdf
[53] Lamping, J.: The interaction of components and aspects. In: Workshop on Aspect Oriented Programming (ECOOP 1997) (1997)
[54] Ossher, H., Tarr, P.: Some micro-reuse challenges. In: Workshop on Advanced Separation of Concerns (ECOOP 2001) (2001)
[55] Vollmann, D.: Visibility of join-points in AOP and implementation languages. In: Costanza, P., Kniesel, G., Mehner, K., Pulvermüller, E., Speck, A. (eds.) Second Workshop on Aspect-Oriented Software Development of the German Information Society, Institut für Informatik III, Universität Bonn (2002); Technical report IAI-TR-2002-1, ISSN 0944-8535
[56] Havinga, W., Nagy, I., Bergmans, L., Aksit, M.: Detecting and resolving ambiguities caused by inter-dependent introductions. In: Proceedings of 5th International Conference on Aspect-Oriented Software Development, AOSD 2006. ACM, New York (2006)
[57] Havinga, W., Nagy, I., Bergmans, L.: An analysis of aspect composition problems. In: Kniesel, G. (ed.) EWAS 2006 – Third European Workshop on Aspects in Software – Proceedings, University of Twente, Enschede, NL, August 31, 2006, pp. 1–8. Technical Report IAI-TR-2006-6, Computer Science Department III, University of Bonn (2006) ISSN 0944-8535
[58] Kastenberg, H., Rensink, A.: Model checking dynamic states in groove. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 299–305. Springer, Heidelberg (2006)
[59] Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfalz, J., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
[60] Durr, P., Bergmans, L., Aksit, M.: Reasoning about behavioural conflicts between aspects. TOASD special issue on Aspect Dependencies and Interactions (submitted) (2006)
[61] Durr, P., Bergmans, L., Aksit, M.: Reasoning about semantic conflicts between aspects. In: Chitchyan, R., Fabry, J., Bergmans, L., Nedos, A., Rensink, A. (eds.) Proceedings of ADI 2006 Aspect, Dependencies, and Interactions Workshop, pp. 10–18. Lancaster University (2006)
[62] Nagy, I., Aksit, M., Bergmans, L.: Composition graphs, a foundation for reasoning about aspect-oriented composition. In: [74]

[63] Nagy, I., Bergmans, L., Aksit, M.: Declarative aspect composition (2004)
[64] Durr, P.: Detecting semantic conflicts between aspects. Master's thesis, University of Twente (2004)
[65] Goldman, M., Katz, S.: Modular generic verification of LTL properties for aspects. In: Mezini, M., Leavens, G.T., Lämmel, R., Clifton, C. (eds.) Foundations of Aspect-Oriented Languages (2006), http://www.cs.iastate.edu/~leavens/FOAL/index-2006.shtml
[66] Katz, S.: Aspect categories and classes of temporal properties. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 106–134. Springer, Heidelberg (2006)
[67] Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
[68] Kniesel, G., Costanza, P., Austermann, M.: JMangler - A Powerful Back-End for Aspect-Oriented Programming. In: Filman, R., Elrad, T., Clarke, S., Akşit, M. (eds.) Aspect-Oriented Software Development, vol. 7. Addison-Wesley, Reading (2004)
[69] Hanenberg, S., Hirschfeld, R., Unland, R.: Morphing aspects: incompletely woven aspects and continuous weaving. In: AOSD 2004: Proceedings of the 3rd international conference on Aspect-oriented software development, pp. 46–55. ACM Press, New York (2004)
[70] Hannemann, J., Chitchyan, R., Rashid, A. (eds.): Workshop on Analysis of Aspect-Oriented Software (AAOS) at ECOOP 2003 (2003), http://www.st.informatik.tu-darmstadt.de:8080/ecoop/workshops/16.phtml
[71] Bachmendo, B., Hanenberg, S., Herrmann, S., Kniesel, G. (eds.): 3rd Workshop on Aspect-Oriented Software Development (AOSD-GI) of the SIG Object-Oriented Software Development. German Informatics Society (2003)
[72] FOAL 2002: Foundations of Aspect-Oriented Langauges (AOSD 2002) (2002)
[73] Bergmans, L., Brichau, J., Tarr, P., Ernst, E. (eds.): SPLAT: Software engineering Properties of Languages for Aspect Technologies (2003)
[74] Leavens, G.T., Clifton, C. (eds.): FOAL: Foundations of Aspect-Oriented Languages (2003)

Appendix: Basics of First-Order Logic

Our approach is based on first-order logic (FOL). This appendix provides a summary of FOL notions and notations used in this paper. A *first-order logic language* consists of predicate symbols, function symbols, constants, variables, and the logic symbols $\wedge, \vee, \neg, \Rightarrow, =, \forall, \exists$. A *formula* is an atom or the connection of formulas by the logic symbols $\wedge, \vee, \neg, \Rightarrow$. An *atom* is an n-ary predicate symbol applied to n argument terms. A *literal* is a positive or negated atom. A *term* is either a constant, a variable (indicated by italics and an initial capital letter, e.g. *Class*), or a compound term. A compound term is an n-ary function symbol applied to n argument terms, e.g. $f(1, X, g(Z))$.

The *Herbrand universe* of a language is the set of all the terms that can be built from its function symbols and constants. The *Herbrand base* is the set of all ground atoms that can be formed from predicate symbols and terms from the Herbrand universe. A *ground* term (or atom) is one that contains no variables.

A *substitution* specifies a replacement of variables by terms. Substitutions are denoted here by lower case Greek letters θ, γ, and substitution sets by upper case letters, Θ, Γ. Application of a substitution $\theta = [V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n]$ to a term t, written $t\theta$, replaces all the occurrences of V_i in t by $t_i\theta$ for $i = 1...n$. For instance, $f(X, 1)[Y \leftarrow 2, X \leftarrow g(Y)] = f(g(2), 1)$. A renaming substitution is one where $t_1, ..., t_n$ are variables distinct from $v_1, ..., v_n$. A *ground* substitution is one where $t_1\theta, \ldots, t_n\theta$ are variable free.

A *unifier* is a substitution that makes two or more terms equal. If θ is a unifier for a set of terms containing t and for every other unifier θ' of the same set there is a substitution γ that is not a renaming substitution such that $t\theta\gamma = t\theta'$ then θ is a *most general unifier*. We write $\theta = \text{mgu}(t_1, t_2)$ to express that t_1 is unifiable with t_2 by the most general unifier θ.

AspectOptima: A Case Study on Aspect Dependencies and Interactions

Jörg Kienzle, Ekwa Duala-Ekoko, and Samuel Gélineau

School of Computer Science, McGill University,
Montreal, QC, Canada H3A 2A7
{Joerg.Kienzle@,{Ekwa.Duala-Ekoko,Samuel.Gelineau}@mail.}mcgill.ca

Abstract. This paper presents AspectOptima, a language independent, aspect-oriented framework consisting of a set of ten base aspects—each one providing a well-defined reusable functionality—that can be configured to ensure the ACID properties (Atomicity, Consistency, Isolation, and Durability) for transactional objects. The overall goal of AspectOptima is to serve as a case study for aspect-oriented software development, particularly for evaluating the expressivity of AOP languages and how they address complex aspect interactions and dependencies. The ten base aspects of AspectOptima are simple, yet have complex dependencies and interactions among each other. To implement different concurrency control and recovery strategies, these aspects can be composed and assembled into different configurations; some aspects conflict with each other, others have to adapt their run time behavior according to the presence or absence of other aspects. The design of AspectOptima highlights the need for a set of key language features required for implementing reusable aspect-oriented frameworks. To illustrate the usefulness of AspectOptima as a means for evaluating programming language features, an implementation of AspectOptima in *AspectJ* is presented. The experiment reveals that *AspectJ*'s language features do not directly support implementation of reusable aspect-oriented frameworks with complex dependencies and interactions. The encountered *AspectJ* language limitations are discussed, workaround solutions are shown, potential language improvements are proposed where appropriate, and some preliminary measurements are presented that highlight the performance impact of certain language features.

Keywords: aspect dependencies, aspect collaboration, aspect interference, reusability, aspect-oriented language features, aspect binding, inter-aspect ordering, inter-aspect configurability, per-object aspects, dynamic aspects.

1 Introduction

Aspect orientation [1] has been accepted as a powerful technique for modularizing crosscutting concerns during software development in so-called *aspects*. Research

A. Rashid and H. Ossher (Eds.): Transactions on AOSD V, LNCS 5490, pp. 187–234, 2009.

has shown that aspect-oriented programming (AOP) is successful in modularizing even very application-independent, general concerns such as distribution [2], concurrency [3, 4], persistency [2, 5], and failures [4].

However, the complexity of aspect-oriented software development increases exponentially when aspects are used in combination with each other. Dependencies between aspects raise the question of how to express aspect configurations, aspect interactions, conflicts among aspects, and aspect ordering. One way to find answers to these questions is to investigate non-trivial and realistic case studies, in which an application with many different concerns is implemented using many interacting aspects.

To this intent we designed AspectOptima [6], a language independent, aspect-oriented framework consisting of a set of ten base aspects—each one providing a well-defined reusable functionality—that can be assembled in different configurations to ensure the ACID properties (Atomicity, Consistency, Isolation, and Durability) for transactional objects. The primary objective of AspectOptima is to provide the aspect-oriented research community with a language independent framework that can serve as a case study for evaluating the expressivity of AOP languages, the performance of AOP environments, and the suitability of aspect-oriented modeling notations, aspect-oriented testing techniques and aspect-oriented software development processes. The aspects that make up AspectOptima are simple, yet have complex dependencies and interactions among each other. The aspects can be composed and assembled into different configurations to achieve various concurrency control and recovery strategies. This composition is non-trivial; some aspects conflict with each other, others cannot function without the support of other aspects, and others have to adapt their run time behavior according to the presence or absence of other aspects.

We believe that studies such as this one are essential to discover key language features for dealing with aspect dependencies and interactions. The experience gained allows researchers to evolve aspect-oriented languages to even better modularize crosscutting concerns, reason about composition, and, most importantly, provide powerful and elegant ways of reuse. The second part of the paper demonstrates this by showing an implementation of AspectOptima in *AspectJ*. The goal of this effort was not to implement AspectOptima in an elegant way using the most appropriate AOP language. The idea was rather to show how the exercise of implementing AspectOptima can highlight the elegance or the lack of programming language features that can appropriately address aspect dependencies and interference in a reusable way. We have chosen to perform our implementation in *AspectJ* since it is currently one of the most popular and stable AOP languages, and *not* because of the features it provides.

The paper is structured as follows: Section 2 describes the context of the case study, namely transactional systems, the ACID properties of transactional objects, concurrency control, and recovery strategies. We present the design of AspectOptima in Sect. 3; it details the design of the 10 well-defined, reusable base aspects, and provides a description of three aspects that implement various concurrency control and recovery strategies by composing the base aspects in

different ways. This design highlights the need for a set of key language features required for implementing reusable aspect-oriented frameworks. In Sect. 4, we describe how we used AspectOptima to evaluate the expressiveness of the language features offered by the AOP language *AspectJ*. We present parts of our implementation to demonstrate that *AspectJ* provides sufficient (but certainly not ideal or elegant) support for dealing with mutually dependent and interfering aspects, discuss the encountered language limitations, suggest possible language improvements where appropriate, and present some preliminary performance measurements. Section 5 comments on related work and the last two sections present the conclusions and future work.

2 Background on Transactional Systems

We present the context of our case study in this section of the paper. The concepts of transactional systems, concurrency control, and recovery strategies relevant to this work are presented in Sects. 2.1, 2.2, and 2.3, respectively.

2.1 Transactional Objects

A *transaction* [7] groups together an arbitrary number of operations on *transactional objects*, which encapsulate application data, ensuring that the effects of the operations appear indivisible with respect to other concurrent transactions. The transaction scheme relies on three standard operations: *begin*, *commit*, and *abort*, which mark the boundaries of a transaction. After beginning a new transaction, all update operations on transactional objects are done on behalf of that transaction. At any time during the execution of the transaction it can *abort*, which means that the state of all accessed transactional objects is restored to the state at the beginning of the transaction (also called *rollback*). Once a transaction has completed successfully (is *committed*), the effects become permanent and visible to the outside.

Classic transaction models typically assume that each transaction is executed by a single thread of control. In recent years, however, advanced multithreaded transaction models have been proposed. Our case study implements the *Open Multithreaded Transaction* model [8, 9], which allows several *participants* (threads or processes) to enter the same transaction in order to perform a joint activity.

The ACID Properties. Transactions focus on preserving and guaranteeing important properties of application data, assuming that all the information is properly encapsulated inside transactional objects. These properties are referred to as the *ACID properties*: Atomicity, Consistency, Isolation, and Durability [7].

Atomicity ensures that from the outside of a transaction, the execution of the transaction appears to jump from the initial state to the result state, without any observable intermediate state. Alternatively, if the transaction cannot be completed for some reason, it appears as though it had never left the initial state.

This *all-or-nothing* property is unconditional, i.e., it holds whether the transaction, the entire application, the operating system, or any other components function normally, function abnormally, or crash.

The *consistency* property states that the application data always fulfill the validity constraints of the application. To achieve this property, transaction systems rely on the application programmer to write *consistency preserving* transactions. In this case, a transaction starts with a consistent state, and recreates that consistency after making its modifications, provided it runs to completion. The transaction system guarantees only that the execution of a transaction will not erroneously corrupt the application state.

The *isolation* property states that transactions that execute concurrently do not affect each other. In other words, no information is allowed to cross the boundary of a transaction until the transaction completes successfully.

Durability guarantees that the results of a committed transaction remain available in the future: the system must be able to re-establish a transaction's results after any type of subsequent failure.

Atomicity and isolation together result in transaction *serializability* [10], guaranteeing that any result produced by a concurrent execution of a set of transactions could have been produced by executing the same set of transactions serially, i.e., one after the other, in some arbitrary order.

When a participant calls a method on a transactional object, the underlying transaction support must take control and perform certain actions to ensure that the ACID properties can be guaranteed. Traditionally, this activity has been divided into concurrency control and recovery activities, which are described in the next two sections.

2.2 Concurrency Control

Concurrency control is that part of the transaction runtime that guarantees the isolation property for concurrently executing transactions, while preserving data consistency. In order to perform concurrency control, conflicting operations, i.e., operations that could jeopardize transaction serializability when executed by different transactions, have to be identified.

The simplest form of concurrency control among operations of a transactional object is *strict concurrency control*. It distinguishes *read*, *write*, and *update* operations. Reading a value from a data structure does not modify its contents, writing a value to the data structure does. Reading and subsequently writing a data structure is called updating. The conflict table of read, write, and update operations is shown in Table 1. There exist more advanced techniques of constructing a conflict table that take into account the semantics of operations [11], but they are out of the scope of this paper.

At run time, concurrency control is performed by associating a concurrency manager with each transactional object. The manager uses the conflict table to isolate transactions from each other. This can be done in a *pessimistic* (*conservative*) or *optimistic* (*aggressive*) way, both having advantages and disadvantages.

Table 1. Strict Concurrency Control Conflict Table

	read	***write/update***
read	No	Yes
write/update	Yes	Yes

Pessimistic Concurrency Control. The principle underlying *pessimistic concurrency control* schemes [7] is that, before attempting to perform an operation on any transactional object, a transaction has to get permission to do so from the concurrency control manager. The manager then checks if the operation would create a conflict with other uncommitted operations already executed on the object on behalf of other transactions. If so, the calling transaction is blocked or aborted.

Optimistic Concurrency Control. With *optimistic concurrency control* schemes [12], transactions are allowed to perform conflicting operations on objects without being blocked, but when they attempt to commit, the transactions are validated to ensure that they preserve serializability. If a transaction passes validation successfully, it means that it has not executed operations that conflict with operations of other concurrent transactions. It can then commit safely. A distinction can be made between optimistic concurrency control schemes based on the validation technique used in determining conflicts. *Forward validation* ensures that committing transactions do not invalidate the results of other transactions still in progress. *Backward validation* ensures that the result of a committing transaction has not been invalidated by recently committed transactions.

Concurrency Control and Versioning. In order to avoid rejecting operations that arrive out of order, several concurrency protocols have been proposed that maintain multiple *versions* (copies of the state) of objects [13–15]. For each update or write operation on an object, a new version of the object is produced. Read operations are performed on an appropriate, old version of the object, thereby minimizing the interactions between read-only transactions and write/update transactions. Versions are transparent to transactions: objects appear to them as only having a single state.

Concurrency Control and Multithreaded Transactions. Advanced transaction models such as Open Multithreaded Transactions allow several threads to perform work within the same transaction. These threads should not be isolated from each other. On the contrary, they should be allowed to see each other's state changes. However, concurrency control must still guarantee data consistency by ensuring that all modifications are performed in mutual exclusion.

2.3 Recovery

Recovery takes care of atomicity and durability of state changes made to transactional objects by transactions in spite of sophisticated caching techniques and

system failures. In other words, recovery must make sure that either all modifications made on behalf of a committing transaction are reflected in the state of the accessed transactional objects or none is, which means that any partial execution of state modifying operations has to be undone.

Recovery actions have to be taken in two situations: on *transaction abort* and in case of a *system failure*. Recovery strategies are either based on *undo* operations or *redo* operations, or both. Supporting both operations allows the cache management to be more flexible.

In order to be able to recover from a system failure, the recovery support must keep track of the status of all running transactions and of the modifications that the participants of a transaction have made to the state of transactional objects. Depending upon the meta-data available about the performed operations, whether a simple read/write classification or a more thorough semantic description, different kind of information will need to be kept. This so-called *transaction trace* must be stored in a log, i.e., on a stable storage device [16] that is not affected by system failures. Once the system restarts, the recovery support can consult the log and perform the cleanup actions necessary to restore the system to a consistent state.

In-place Update and Deferred Update. There exist two different strategies for performing updates and recovery for transactional objects, namely *in-place update* and *deferred update*.

When using *in-place update*, all operations are executed on *one main copy* of a transactional object. The effects of the operation invocation are therefore potentially visible to all following operation invocations, even those made on behalf of other transactions. In order to be able to undo changes in case of transaction abort, a backup copy, *snapshot* or *checkpoint* of an object is made before a transaction modifies the object's state.

When using *deferred update*, each modifying transaction executes operations on a *different copy* of the state of a transactional object. Therefore, until it commits, a transaction's changes are not visible to other transactions. When it does commit, the effects of its operations are applied to the original object either by explicitly copying the state or by reapplying the transaction's operations on the main copy.

2.4 Putting Things Together

In order to guarantee the ACID properties, each time a method is invoked on a transactional object the following actions must be taken:

1. *Concurrency Control Prologue*: The concurrency control associated with the transactional object has to be notified of the method invocation to come. Pessimistic concurrency control schemes will use this opportunity to block or abort the calling transaction in case of conflicts.
2. *Recovery Prologue*: The recovery manager has to be notified in order to collect information for undoing the method call in case the transaction aborts later on.

3. *Method Execution*: The actual method call is executed.
4. *Recovery Epilogue*: The recovery manager has to be notified to gather redo information, if necessary.
5. *Concurrency Control Epilogue*: The concurrency control has to be notified that the method execution is now over.

Every transaction has to also remember all accessed transactional objects. When a transaction commits or aborts, it has to notify the concurrency control and recovery managers of each object of the transaction outcome.

3 The AspectOptima Framework

As shown in [4], transactions cannot be transparently added to an application. Yet, once the programmer has decided to use transactions in his application, and decided upon transaction boundaries, and determined the state that has to be encapsulated in transactional objects, it is possible to provide a middleware/framework/library that provides the run time support for transactions. Optima (OPen Transaction Integration for Multi-threaded Applications) [9, 17] is an object-oriented framework providing such support.

In this paper, we present the design of AspectOptima, a purely aspect-oriented framework ensuring the ACID properties for transactional objects. We present the design rationale of our framework in Sect. 3.1; Section 3.2 describes each of the ten base aspects of AspectOptima and Sect. 3.3 describes how these aspects can be combined to implement different concurrency control and recovery strategies for transactional objects.

3.1 Design Rationale

At a higher level, concurrency control and recovery can be seen as two completely separate concerns. As explained in the previous section, there are different ways of performing concurrency control and recovery, and, depending on the application, a developer might want to choose one technique over the other to maximize performance. Based on our experience of implementing the object-oriented Optima, we know that, at the *implementation* level, concurrency control and recovery cannot be separated completely from each other. There may be conflicts between the two, since not all combinations of concurrency control and recovery techniques mentioned before successfully provide the ACID properties. For example, most optimistic concurrency control techniques do not work with in-place update. There is also overlap between the two, since both techniques have to gather similar run time information in order to correctly perform concurrency control and recovery. For example, they both need to distinguish read from write/update operations.

Motivated by this incomplete separation of concerns, we applied aspect-oriented reasoning to decompose concurrency control and recovery further, and identified a set of ten aspects, each providing a specific sub-functionality. We

did not follow any particular aspect-oriented design technique to determine the functionality and scope of each aspect. Instead we relied on our object-oriented experience in implementing the ACID properties to identify and modularize reusable crosscutting functionality within aspects. Each of these aspects provides a well-defined common functionality, which, as it turns out, is often needed in non-transactional applications as well. In order to allow the aspects to be used in other contexts, they have been carefully designed to be reusable.

3.2 Ten Individually Reusable Aspects

This section describes each of the ten base aspects of AspectOptima. The motivation for each aspect is given, together with a brief summary of its functionality. Then the dependencies of the aspect are listed, i.e., what other base aspects the current aspect depends on, and to what other aspects the current aspect provides essential functionality. Situations of aspect interference, i.e., aspects that have to modify their behavior in order to continue to provide their functionality in the presence of other aspects, are pointed out. Finally, at the end of each aspect description, we sketch examples of how the aspect could be used in a stand-alone way to convince the reader of its reuse potential.

AccessClassified

Motivation. Concurrency control and recovery can be done more efficiently if the operations of transactional objects are classified according to how they affect the object's state: *read operations* (observers)—operations that do not modify the state of an object, *write operations,* and *update operations* (modifiers)—operations that read and write the state of an object.

The *AccessClassified* aspect provides this classification for methods of an object. Ideally, the classification of the operations should be done automatically. However, some implementations might require assistance from the developer. In such a case, the aspect should detect obvious misclassifications and output warnings.

Summary of Functionality of AccessClassified

- Every method of the object must be classified as either a *read*, *write*, or *update* operation. This functionality can, for instance, be provided by an operation `Kind getKind(Method m)`.

Dependencies of AccessClassified

- *Depends on*: –
- *Interferes with*: –
- *Is used by*: *Shared*, *Tracked, AutoRecoverable,* Concurrency Control, Recovery

Reusability of AccessClassified. The information provided by *AccessClassified* is very useful in many contexts. It could be exploited in multi-processor systems to implement intelligent caching strategies, or help to choose among different communication algorithms and replication strategies in distributed systems.

Named

Motivation. One of the fundamental properties of an object is its *identity*. Identity is the property that distinguishes an object from all other objects. It makes the object unique. At run time, a reference pointing to the memory location at which the state of an object is stored is often used to uniquely identify an object.

The lifetime of transactional objects, however, is not linked to the lifetime of an application, even less to a specific memory location. Since the state of a transactional object survives program termination, there must be a unique way of identifying a transactional object that remains valid during several executions of the same program. This identification means must allow the run time support to retrieve the object's state from a storage device or database. Also, depending on the chosen concurrency control and recovery techniques, there might exist multiple copies of the state of a transactional object in memory at a given time. These multiple copies, however, represent in fact one application object.

Previous work [18] has shown that an object *name* in the form of a string is an elegant solution for uniform object identification. We can therefore define a *Named* property or aspect for objects. A named object has a name associated with its identity. A name must be given to the object at creation time. The name remains valid throughout the entire lifetime of the object. At any time it should be possible to obtain the name of a given object, or retrieve the object based on its name.

Summary of Functionality of Named

- All creator operations must associate a unique name with the object that is created.
- `Object getObject(String s)` and `String getName(Object o)` operations map from an object to a name and vice versa.

Dependencies of Named

- *Depends on*: –
- *Interferes with*: –
- *Is used by*: *Tracked*, *Persistent*

Reusability of Named. Named can be used as a stand-alone aspect whenever the logical lifetime of an object does not coincide with the actual lifetime of its pointer into memory. For instance, one might like to destroy large, referenced but empirically unused objects and recompute them if needed. A name can also be used as a key to retrieve the state of an object from a database.

Shared

Motivation. Transactional objects are shared data structures. To conserve data consistency, it is important to prevent threads from concurrently modifying an object's state. Such a situation could arise when participants *of the same transaction* want to simultaneously access the same object.[1]

The *Shared* aspect ensures multiple readers/single writer access to objects—all modifications made to the state of a shared object are performed in mutual exclusion.

Summary of Functionality of Shared

- Before executing the method body of a shared object, a read lock or write lock has to be acquired.
- When returning from the method, the previously acquired lock has to be released.

Dependencies of Shared

- *Depends on*: *AccessClassified*
- *Interferes with*: –
- *Is used by*: Concurrency Control, Recovery

In order to maximize throughput and hence optimize performance, *Shared* needs the semantic information of each method of the object provided by *AccessClassified*. If this information is not available, *Shared* must make worst case assumptions, i.e., assume that all methods of the object are in fact write or update operations.

All concurrency control schemes rely on *Shared* to provide consistency of data updates in spite of simultaneous accesses made by participants of the same transaction.

Reusability of Shared. Shared can be used as a stand-alone aspect in any concurrent application to guarantee data consistency of shared objects.

Copyable

Motivation. An object encapsulates state. The state of an object is initialized at creation time and can be altered by each method invocation. In a sense, the state of an object is its memory.

Sometimes it is necessary to duplicate an object, or replace an object's state with the state from another object. This functionality is offered by *Copyable*.

[1] The functionality offered by *Shared* is *not* to provide isolation among threads running in *different* transactions, but mutual exclusion among threads *of the same transaction*. This functionality is transaction specific and can be implemented in many different ways. We will show later how isolation can be achieved by combining several of the ten base aspects, such as demonstrated in *LockBased*, *MultiVersion*, and *Optimistic* (see Sect. 3.3).

Summary of Functionality of Copyable

- `Object clone()` creates an identical copy of the object.
- `replaceState(source)` copies the state of `source` over the state of the current object.

There is no obvious answer to the question whether to perform a *deep copy* or a *shallow copy* of the state of an object. When an object A stores in its state a reference to an object B, deep copy also clones/replaces the state of B when cloning/replacing the state of A. Recursively, if B refers to other objects, they are cloned/replaced as well. Shallow copy, on the other hand, only clones/replaces the state of A. Which technique is ideal depends on the application. Sometimes an application might even want to handle different classes/objects in different ways. A flexible implementation of *Copyable* should therefore provide a default technique, but allow the user to override the default behavior if needed.

Dependencies of Copyable

- *Depends on*: –
- *Interferes with*: *Shared*
- *Is used by*: *Serializable*, *Versioned*

Copyable should detect the presence of *Shared*. In a multi-threaded environment, the state of a shared object can only be copied when no other thread is modifying it.

Reusability of Copyable. *Copyable* is used whenever an object's state must be copied into another object of the same class. This situation arises whenever an object needs to be duplicated, e.g. for caching or replication. It is so often encountered that most programming languages provide the functionality of *Copyable* within the language (e.g. the Java `Object clone()` method that can be invoked on all objects that implement the `Cloneable` interface).

Serializable

Motivation. When an object is created, its state is in general stored in main memory. Whenever the object's state has to be moved to a different location, e.g. to a file, a database, or over the network to a different machine, the in-memory representation of the state of the object has to be transformed to suit the representation required by the destination location.

Serializable provides this functionality. A serializable object knows how to read its state from and write its state to backends requiring varying data representation formats. It is an incarnation of the *Serializer* pattern described in [19]. Just like with *Copyable*, serialization can be deep or shallow. Again, the ideal way of performing serialization is application-dependent. *Serializable* should therefore be customizable to specific application needs.

Summary of Functionality of Serializable

- `readFrom(reader)` restores the state of the object by reading it from a back end.
- `createFrom(reader)` creates a new object and initializes it with the state read from a back end.
- `writeTo(writer)` saves the state of the object to a back end.

Dependencies of Serializable

- *Depends on*: *Copyable*
- *Interferes with*: *Shared, Named*
- *Is used by*: *Persistent*

Serializable should detect the presence of *Shared*. In a multi-threaded environment, a shared object should only be serialized when no other thread is modifying it. *Serializable* should also detect the presence of *Named*, and serialize the name together with the object's state.

Reusability of Serializable. Serializable can be used in many situations, e.g., for writing an object's state to a file, sending the state over the network, or displaying an object's state. Serialization is so handy that modern programming languages usually provide a default serialization implementation for objects. The default serialization can usually be overridden with customized serialization, if needed.

Versioned

Motivation. During execution, each transaction must have its own view of the set of objects it accesses. Every thread participating in a transaction should see updates made by other participants, but not updates made from within other transactions. This is why multi-version concurrency control strategies, as well as snapshot-based recovery techniques, have to create multiple copies of the state of a transactional object in order to isolate state changes made by different transactions from each other.

This functionality is provided by *Versioned*. A *Versioned* object can encapsulate multiple copies—*versions*—of its state. Versions are linked to *views*, one of which is the default *main view*. If a *main view* is not explicitly designated, the original state of the object when it was created becomes the default view. A thread can subscribe to a view, and any method call made subsequently by the thread is directed to the associated version. A call coming from a thread that is not part of a specific view is forwarded to the main view.

Summary of Functionality of Versioned

- `Version newVersion()` creates a new version. The returned *Version* object is a "handle" to the newly created version.
- `deleteVersion(Version v)` deletes the version *v*.

- `View newView()`, `joinView(View v)`, `leaveView()`, and `deleteView(View v)` allow a thread to create, join, leave, or delete a view. Views are *nestable*, meaning that if a thread joins view A and then later creates a new view B, and then leaves view B, it should end up back in view A.
- `Version getCurrentVersion()` queries the current version assigned to the view of the calling thread.
- `setCurrentVersion(Version v)` assigns the version v to the view of the calling thread.
- `setMainView(View v)` designates a view to be considered the main one.
- Any method invocation on the transactional object are directed either to the version of a particular view, if the calling thread has joined a specific view, or else by default to the main view.

Dependencies of Versioned

- *Depends on*: *Copyable*
- *Interferes with*: –
- *Is used by*: *Recoverable,* Concurrency Control

Versioned requires the presence of *Copyable* in order to duplicate the object's state when a version is created. *Versioned* interferes with *Shared*: only when no other thread is currently modifying the object's state a new version can be created. Fortunately, *Copyable* should take care of this interference already, and therefore the interference between *Versioned* and *Shared* is only indirect.

Versioned is used by multi-version and optimistic concurrency control schemes, as well as by *Recoverable* for snapshot-based recovery. The optimistic concurrency control presented in Sect. 3.3, for instance, updates the main version of a *Versioned* object after successful validation of a committing transaction and deletes all non-main versions created by this transaction.

Reusability of Versioned. Versioned can be used as a stand-alone aspect in any application requiring transparent handling of multiple instances of an object's state. For instance, if an editor is extended to support different views for a single document, *Versioned* can be used to make different instances of the toolbar act on the correct view even though the toolbar source code refers to a single global variable.

Tracked

Motivation. To guarantee the ACID properties, the transaction runtime has to keep track of state access. The *Tracked* aspect provides the functionality to monitor object access in a generic way. It allows a thread to define a region in which object accesses are monitored. The region is delimited by *begin* and *end operations*. At any given time, the thread can obtain all read or modified objects for the current region.

Summary of Functionality of Tracked

- `TrackedZone beginTrackedZone()`, `joinTrackedZone(TrackedZone z)`, `leaveTrackedZone()`, and `endTrackedZone()` operations are provided that delimit the regions in which object access is to be monitored. Just like views, zones should be *nestable*. When a thread joins zone A, and then later zone B, and then leaves B, it should end up back in A.
- All accesses to a *Tracked* object from within a zone is monitored.
- `Object[] getAccessedObjects()`, `Object[] getReadObjects()`, and `Object[] getModifiedObjects()` operations that return the set of accessed, read, or modified objects of the current zone. Note that if nesting of zones is supported, then read or modified objects of a child zone have to be included when returning the set of read or modified objects of the parent.

Dependencies of Tracked

- *Depends on*: *AccessClassified, Named*
- *Interferes with*: *Versioned*
- *Is used by*: Concurrency Control, Recovery

In order to distinguish observer and modifier methods, *Tracked* depends on the presence of *AccessClassified*. Since it is not necessary to track accesses to different copies of the same application object, *Tracked* should detect the presence of *Versioned*. Using *Named* it is possible to compare the object names instead of object references to avoid duplicate counting.

Tracked is used by the transaction support run time to notify the concurrency controls of all accessed objects when a transaction commits or aborts. The recovery manager uses *Tracked* to identify all objects whose state has been modified and therefore must be made persistent (or rolled back in case of an abort). In this case, the tracked zone begins when the transaction begins, and ends when the transaction ends.

Reusability of Tracked. *Tracked* can be used in a stand-alone way to monitor object access made by arbitrary pieces of code, for instance, for the sake of logging and debugging. As another example, an implementation of the model-view-controller pattern [20] could use *Tracked* to maintain a dirty list of the parts of the model which changed since the view was last redrawn.

Recoverable

Motivation. The transaction runtime must be able to undo state changes made on behalf of a transaction when it aborts. *Recoverable* provides that functionality.

A recoverable object [21] is an object whose state can be saved and later on restored, if needed. Saving is sometimes also referred to as establishing a checkpoint; it is usually performed when the object is in a consistent state. Once saved, the state of the object can be restored at any time. It is possible to establish multiple checkpoints of the state of an object.

To implement checkpointing, this version of *Recoverable* takes a snapshot of the state of an object. In the future we might want to support logical recovery as well (see Sect. 7).

Recoverable should support both in-place and deferred update strategies (see Sect. 2.3). To support nested transactions it must be possible to establish multiple checkpoints for a single object. In case of in-place update, this creates a "sequence" of checkpoints, i.e., each checkpoint has at a given time at most one predecessor and one follower. In case of deferred update, a "tree" of checkpoints is created.

Summary of Functionality of Recoverable

- `Checkpoint establishCheckpoint()` creates a checkpoint. Depending on the chosen update strategy, *Recoverable* either makes sure that all views continue to point to the original copy of the object (in-place update) or creates a new version of the object and assigns it only to the view associated with the calling thread (deferred-update).
- `restoreCheckpoint(Checkpoint c)` restores the object's state to the state of a previously established checkpoint *c*. If no checkpoint is given as a parameter, then strict nesting of checkpoints is assumed, and the latest checkpoint is discarded.
- `discardCheckpoint(Checkpoint c)` discards a checkpoint *c*. If no checkpoint is given as a parameter, then strict nesting of checkpoints is assumed, and the latest checkpoint is discarded.
- `setDeferred(boolean On)` switches between in-place and deferred-update mode.

Dependencies of Recoverable

- *Depends on*: *Versioned*
- *Interferes with*: –
- *Is used by*: *AutoRecoverable,* Recovery

Recoverable depends on *Versioned* to provide snapshot-based checkpointing. It indirectly interferes with *Shared*: only when no other thread is currently modifying the object's state, a checkpoint should be established. Fortunately, *Versioned* should take care of this interference already.

Reusability of Recoverable. *Recoverable* can be used in any application that wants to be able to recover a previous state of an object. For instance, it can be used to implement a simple undo functionality.

AutoRecoverable

Motivation. In order to be able to rollback the state of an application when a transaction aborts, all accesses to transactional objects have to be monitored, and when an object is going to be modified for the first time from within the transaction, a checkpoint has to be established.

The *AutoRecoverable* aspect provides such region-based recovery. It allows a thread to define a region in which object access is monitored. The region is delimited by *begin* and *end operations*. Within a region, before any modifications are made to an object's state, a checkpoint is established automatically.

Summary of Functionality of AutoRecoverable

- `beginRecoverableZone()`, `joinRecoverableZone(RecoverableZone z)`, `leaveRecoverableZone()`, and `endRecoverableZone()` operations that delimit the regions in which object access is to be monitored for the current thread. Zones should be nestable.
- Whenever an auto-recoverable object is modified for the first time from within a zone, a checkpoint of the object has to be established.

Dependencies of AutoRecoverable

- *Depends on*: *Recoverable, AccessClassified*
- *Interferes with*: –
- *Is used by*: Recovery

AutoRecoverable depends on *Recoverable* to provide undo functionality for the object. It also depends on *AccessClassified* to determine if an operation is modifying the object's state or not.

Reusability of AutoRecoverable. AutoRecoverable can be used in any application that wants to be able to recover state changes made by arbitrary pieces of code. For instance, it can be used to undo the operations of a user-defined command manipulating an unknown subset of the application objects.

Persistent

Motivation. Persistent objects are objects whose state survives program termination. To achieve this, persistent objects know how to write their state to stable storage [16], i.e., a reliable secondary storage such as a mirrored hard disk or a database. Subsequently, it is possible to reinitialize the object's state based on the content of the storage device.

Summary of Functionality of Persistent

- All creator operations (constructors) of the object must associate a well-defined location on a storage device with the object.
- Operations to load/save the state of the object from/to the associated storage device.
- An operation to destroy a persistent object. When the object ceases to exist, the space on the associated storage device has to be freed as well.

Dependencies of Persistent

- *Depends on*: *Serializable*, *Copyable*, *Named*
- *Interferes with*: *Versioned, Recoverable*
- *Is used by*: Recovery

Persistent requires the presence of *Serializable* in order to transform the object's state into a flat stream of bytes. *Persistent* requires the presence of *Named*. It assumes that the object's name designates a valid location on a secondary storage device. *Persistent* should know how to handle the presence of *Versioned*, since, in general, there is a "main" version that contains the state of the object that is currently considered the correct one. *Persistent* should know how to handle the presence of *Recoverable*. When a recoverable object is made persistent, all checkpoints have to be made persistent as well. *Persistent* indirectly interferes with *Shared*. Only when no other thread is modifying the state of the object, *Persistent* should load or save the state of the object. Fortunately, *Serializable* should take care of this interference already.

Persistent is used by the recovery manager to write the old and new state of a transactional object to stable storage before the transaction commits. It is also used by the recovery manager to write information used to achieve tolerance to crash failures to the system log.

Reusability of Persistent. Persistent can be used in any application where an object's state has to survive program termination and hence has to be stored on some non-volatile storage device. To support many different storage devices, *Persistent* should be used in combination with a persistence framework such as [18].

Comments on Persistent. The *Persistent* aspect on its own only supports *explicit persistence*, i.e., the *Persistent* aspect has to be explicitly applied to every object that is to be made persistent. Also, loading and saving of the state of the object has to be done explicitly by invoking the corresponding method.

In a programming language providing *orthogonal persistence* [22], persistent data are created and used in the same way as non-persistent data. Loading and saving of values does not alter their semantics, and the process is transparent to the application program. Whether or not data should be made persistent is often determined using a technique called *persistence by reachability*. The persistence support designates an object as a persistent root and provides applications with a built-in function for locating it. Any object that is "reachable" from the persistent root, for instance by following pointers, is automatically made persistent. Providing orthogonal persistence and persistence by reachability is out of the scope of the AspectOptima framework.

Dependency and Interaction Summary. Figure 1 shows a UML class diagram that depicts the different relationships among the ten base aspects. Dependencies between aspects are shown on the left and interference between aspects on the right. The aspects that have to intercept calls to objects are stereotyped `<<i>>` (for *interceptors*). They all apply to the same joinpoints, i.e., they have to intercept *all* public method calls to the object they apply to. As mentioned before, the order in which they intercept is important.

The right-hand side of the diagram also shows two UML notes labeled *Concurrency Control* and *Recovery*. These show how the ten base aspects relate to

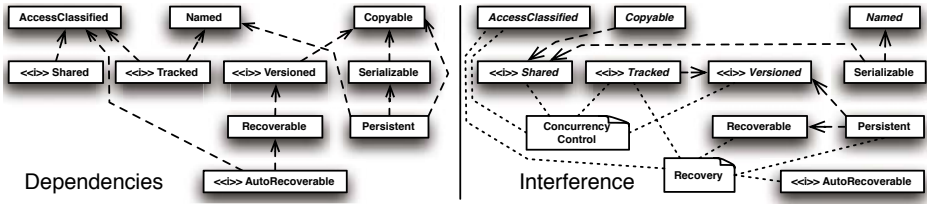


Fig. 1. Aspect Dependencies and Interferences

concurrency control and recovery strategies. The aspects that have italic names are those aspects that are used directly or indirectly by *both* concurrency control and recovery. They represent the implementation overlap between the two high-level concerns mentioned in Sect. 3.1.

3.3 Aspect Compositions

This section describes how the aforementioned base aspects can be combined to implement different concurrency control and recovery strategies for transactional objects. Each of the following aspects requires that all transactional objects are *AccessClassified*, *Named, Copyable*, *Serializable, Shared, Versioned, Tracked, Recoverable*, *AutoRecoverable*, and *Persistent*.[2] The aspects also assume that the transaction runtime creates a tracked zone, a recoverable zone, and a new view when a transaction begins, and ends the tracked zone, the recoverable zone, and the view when a transaction commits or aborts.

Pessimistic Lock-Based Concurrency Control with In-Place Update. This section describes the design of the *LockBased* aspect, which implements pessimistic lock-based concurrency control. Lock-based protocols use locks to implement permissions to perform operations. When a thread invokes an operation on a transactional object on behalf of a transaction, *LockBased* intercepts the call, forcing the thread to obtain the lock associated with the operation. The kind of lock—*read*, *write*, or *update*—is chosen based on the information provided by *AccessClassified*. Before granting the lock, *LockBased* verifies that this new lock does not conflict with a lock held by a different transaction in progress. If a conflict is detected, the thread requesting the lock is blocked and has to wait for the release of the conflicting lock. Otherwise, the lock is granted. *LockBased* then makes sure that *in-place* update has been selected for this object by calling *Recoverable*, and allows the call to proceed.

The order in which locks are granted to transactions imposes an execution ordering on the transactions with respect to their conflicting operations. Two-phase locking [23] ensures serializability by not allowing transactions to acquire any lock after a lock has been released. This implies in practice that a

[2] The functionality provided by *Persistent* is not used in the examples shown in this section. Actually, persistency is mostly required at commit time of a transaction as shown in Sect. 4.3.

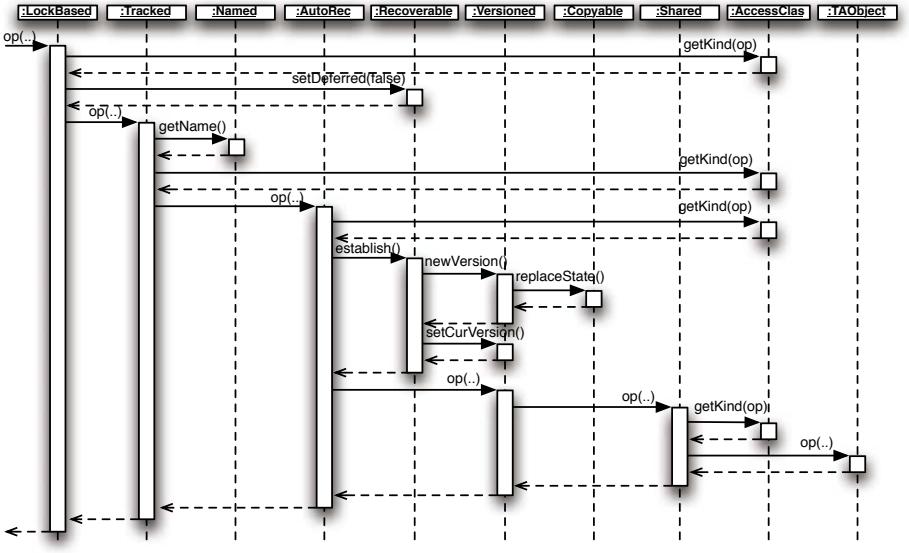


Fig. 2. Aspect Interactions for *LockBased* Objects

transaction acquires locks during its execution (1st phase) and releases them at the end once the outcome of the transaction has been determined (2nd phase).

To release all acquired locks when a transaction ends, all transactional objects that are accessed during a transaction have to be monitored. To this end, *LockBased* depends on *Tracked* to intercept the call and record the access. Obviously, an object should be tracked only after a lock has been granted.

Next, *LockBased* depends on *AutoRecoverable* to intercept the call and to checkpoint the state of the transactional object, if necessary, before it is modified. Since we are using in-place update, *Versioned* then directs the operation call to the main copy of the object. Finally, *Shared* intercepts the call and makes sure that no two threads running in the same transaction are modifying the object's state concurrently.

After the method has been executed, *Shared* releases the mutual exclusion lock. The transactional lock, however, is held until the outcome of the transaction is known. Figure 2 illustrates this interaction; the sequence diagram depicts how a call to a transactional object is intercepted, and how the individual aspects collaborate to provide the desired functionality.

Comments on LockBased. The design of LockBased is currently very simple. It does not support upgrading or downgrading of locks. Also, *LockBased* currently does not detect deadlock. Deadlock situations can happen with any blocking pessimistic concurrency control in case there are circular dependencies between transactions. Deadlock detection can therefore be seen as a crosscutting functionality, and could therefore be added as yet another base aspect to AspectOptima. Starvation is prevented in *LockBased* if the locks are granted in a strict FIFO ordering.

Pessimistic Multi-version Lock-Based Concurrency Control with In-Place Update. One drawback of standard lock-based concurrency control is that *read-only* transactions, i.e., transactions that only invoke observer methods on transactional objects, can be blocked by *update* transactions. This is especially annoying in applications where there are many short-lived read-only transactions, but only a few long-lived update transactions.

The *MultiVersion* aspect addresses this problem by implementing multi-version lock-based concurrency control with in-place update. *MultiVersion* relies on the fact that the transaction runtime knows how to classify transactions into *read-only* transactions and *update* transactions, i.e., transactions that write (and potentially read) the state of an object. *MultiVersion* also assumes that it is possible to assign timestamps with transactions.

MultiVersion keeps multiple versions of the state of a transactional object in memory—the "history" of *committed* states of an object, so to speak. Each version is annotated with a logical time interval during which that state was valid.

Update transactions are handled just as in *LockBased*. First, *MultiVersion* tries to acquire a write lock on the object. If no other transaction is currently modifying the object's state, then the lock is granted; otherwise, the calling thread is suspended. Once the lock is granted, *MultiVersion* relies on *Tracked* to record the access. If this is the first write performed on behalf of the transaction, *AutoRecoverable* checkpoints the object's state using in-place update, creating a new version. *Versioned* then directs the call to the new copy of the object, and finally *Shared* intercepts the call and makes sure that no two threads are modifying the object's state concurrently.

After the call has been executed, *Shared* releases the mutual exclusion lock. Future updates performed by the same transaction are automatically directed by *Versioned* to this version.

When an update transaction commits, *MultiVersion* assigns it a new logical timestamp, and adds the new committed state to the history of states, annotated with the new timestamp.

Read-only transactions are handled differently. They are assigned logical timestamps at *creation time*. They do not have to acquire any locks. *MultiVersion* nevertheless has to intercept the call and look at the transaction timestamp. It then finds the version with the highest timestamp that is lower than the transaction timestamp and assigns this version to the view of the transaction using *Versioned*. The call then proceeds to *Tracked,* where the read access is recorded. Then *Versioned* directs the call to the selected version. There is no need for *AutoRecoverable* or *Shared*, since only read requests are directed to this version.

Figure 3 illustrates the control flow through the aspects when a read or update operation is invoked on a transactional object. The versions *old1*, *old2*, *old3*, and *old4* represent previously committed states of the transactional object. Since they are only accessed by read operations, the *Shared* aspect is not needed anymore. To optimize performance, the *Shared* aspect should be removed from the main version as soon as an update transaction commits.

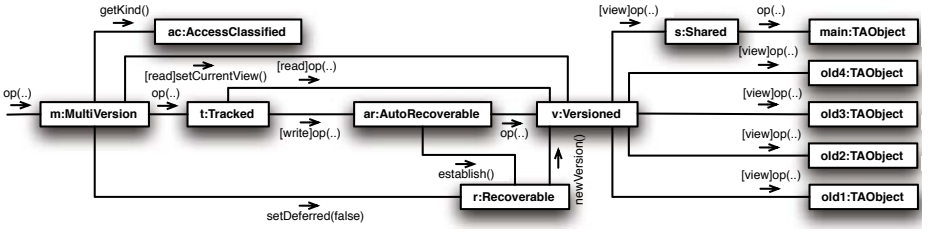


Fig. 3. Control Flow for Multi-Version Concurrency Control

Optimistic Concurrency Control with Deferred Update and Backward Validation. The aspect *Optimistic* implements optimistic concurrency control with deferred update and backward validation. When using optimistic concurrency control, the execution of each transaction is split into a *read phase*, a *validation phase*, and a *write phase*. When a transaction starts, it remembers the timestamp of the most recently committed transaction (*Tstart*). If and only if a transaction passes the validation phase, it receives a timestamp of its own and commits.

During the read phase, the transaction always reads the most recently committed states of transactional objects. *Optimistic* intercepts method calls to transactional objects and queries *AccessClassified* to classify the call.

In case of a read, *Optimistic* passes the call along to *Tracked* to record the read access. *Versioned* forwards the call to the current main version that contains the most recently committed state. There is no need for *AutoRecoverable* to do any work, nor is the presence of *Shared* required, since the call is read-only.

In case of a write or update operation,[3] *Optimistic* makes sure that deferred update is selected by calling *Recoverable*, and then passes the call to *Tracked*. Next, *AutoRecoverable* creates a new version of the transactional object, but this time using deferred update. This ensures that subsequent reads made by other transactions are still forwarded to the most recently committed version. *Versioned* forwards the call to the newly created version, and *Shared* takes care of ensuring mutual exclusion.

In case of a concurrent write made by a different transaction, *AutoRecoverable* creates yet another version. Therefore, at a given time, there might exist *multiple uncommitted versions* of a transactional object, each one belonging to a different transaction.

The UML 2.0 communication diagram shown in Fig. 4 illustrates the control flow through the aspects for read and update operations. In the depicted situation, there are currently four active update transactions, each one having its own local version of the transactional object's state. Read access to the main version does not flow through *AutoRecoverable* or *Shared*.

In optimistic concurrency control schemes, a transaction has to pass *validation* before it can commit. During validation, *Optimistic* looks at the timestamp of the most recently committed transaction *(Tend)*. *Optimistic* then calculates the

[3] Note that we are still in the read phase of the transaction!

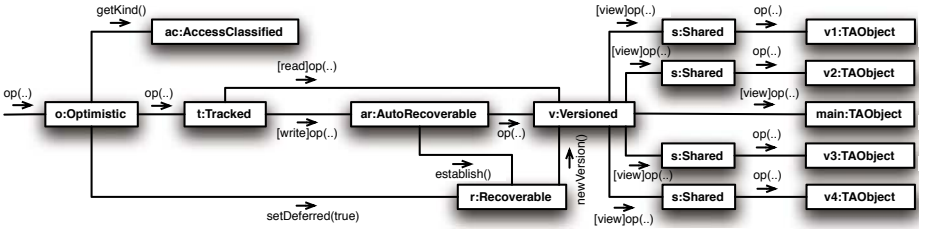


Fig. 4. Control Flow for Optimistic Concurrency Control

union of all transactional objects updated by transactions *Tstart+1* to *Tend* using *Tracked* and intersects it with the set of objects read by the validating transaction (backward validation).

If the intersection is non-empty, validation fails and the transaction has to abort. *Optimistic* tells *Recoverable* to restore the checkpoints of all modified objects, which results in deleting the local versions of the transaction.

If the intersection is empty, validation is successful. The transaction receives a timestamp and proceeds to the write phase, in which *Optimistic* tells *Recoverable* to discard the checkpoints of all modified objects. This results in committing the local versions of the transaction and discarding the previous one.

3.4 Summary

Transaction systems in general implement the ACID properties by performing concurrency control and recovery, each of which can be done using different techniques. This separation of concerns is, however, not very clean. Concurrency control and recovery can benefit from sharing parts of their implementation, and certain combinations of concurrency control techniques conflict with certain recovery techniques.

AspectOptima shows how aspect-oriented techniques can help to decompose the implementation of the ACID properties into a set of fine-grained aspects. The decomposition exhibits the following properties:

- *Clear Separation of Concerns:* Each aspect provides a well-defined functionality. For example, *Shared* takes care of ensuring mutual exclusion of state updates.
- *High Reusability:* Each aspect can be used in other applications in a stand-alone way to implement similar functionalities. For example, *Recoverable* can be used to implement "undo" functionality.
- *Complex Aspect Dependencies:* Some aspects cannot function properly without the functionality offered by other aspects. For example, *Persistent* depends on the presence of *Named*. It uses the object's name to designate a valid location on a secondary storage device.
- *Complex Aspect Interference:* Some aspects have to adapt their functionality in the presence of other aspects. For example, *Copyable* has to detect the presence of *Shared*, and make sure that it only makes a copy of an object when no other thread is modifying the object's state.

4 Evaluating the Expressiveness of AO Languages with AspectOptima

In this section, we demonstrate how AspectOptima can be used to evaluate the expressiveness of the language features of an AOP language, particularly those features concerned with aspect reuse, aspect dependencies and interference. The language under study in our case is *AspectJ* [24]. The goal of this effort was not to implement AspectOptima in an elegant way using the most appropriate AOP language. The idea was rather to show how the exercise of implementing AspectOptima can highlight the elegance or the lack of programming language features that can appropriately address aspect dependencies and interference in a reusable way. We have chosen to perform our implementation in *AspectJ* since it is currently one of the most used languages with a mature and reliable compiler, and *not* because of the features it provides.

This section is structured as follows. We outline the language requirements necessary for implementing AspectOptima in Sect. 4.1; a brief overview of the *AspectJ* is presented in Sect. 4.2. We present an *AspectJ* implementation of AspectOptima in Sect. 4.3, discuss the encountered language limitations of *AspectJ*, present some suggestions for language improvements in Sect. 4.4, and finally show some preliminary measurements that highlight the performance impact of certain language features in Sect. 4.5.

4.1 AspectOptima Implementation Language Requirements

One of the goals of AspectOptima is to define a framework that can be used to evaluate and compare the expressiveness of language features of AOP languages, particularly with respect to how they deal with complex aspect dependencies and interactions in a reusable way. The key questions an implementation has to address are:

- Can each of the aspects be implemented in a *modular*, stand-alone way? To be considered modular, the implementation of an aspect should be packaged in such a way that the package contains all the code needed to implement the functionality. This simplifies adding and removing of an aspect for application developers, and improves readability and maintainability for aspect developers.
- Can each of the aspects be implemented in a *reusable* way? To be considered reusable, a developer that needs a functionality offered by one of the aspects in his application should be able to integrate the functionality with minimal effort into his implementation by simply deploying the aspect.
- Does the AOP language allow to specify different aspect configurations and compositions of the ten base aspects? Can *different* combinations be used within the *same* application? In aspect frameworks it is likely that different aspect combinations are possible, and could be useful at different places in the application.

- Is aspect configuration *safe*? Aspect deployment should not be error-prone, i.e., it should not happen that an application developer can make mistakes when deploying the aspect in his application.
- Can *dependencies* between aspects be handled in a *transparent* way? To be considered transparent, a developer that needs a functionality offered by one aspect should not have to explicitly deal with aspect dependencies, i.e., when deploying an aspect A, all aspects that A depends on should be automatically deployed as well.
- Can *interferences* between aspects be dealt with in such a way that the aspect implementations are still individually reusable, i.e., there are no direct dependencies among the aspect implementations due to aspect interference? When two aspects interfere and additional behavior is required to address the interference, can this be dealt with in a transparent way without bothering the application developer at configuration time?

Based on our experience, the following list summarizes what features an AOP language and environment has to offer to make the implementation of AspectOptima possible:

- *Separate Aspect Binding:* In order to support reusability, reusable aspect implementations should not contain explicit bindings to application elements. An application developer has to be able to specify where an aspect is to be applied when he composes his application.
- *Inter-Aspect Configurability:* Some aspects have to be able to express their dependence on other aspects. For example, *Versioned* can only be applied to objects that are also *Copyable*.
- *Inter-Aspect Ordering:* Some aspects need to specify the order in which other aspects get applied. For example, the aspect *LockBased* has to make sure that *Tracked* records the object access only after a lock has been acquired.
- *Per-Object (Per-Instance) Aspects:* An application programmer might want to use different concurrency control or recovery implementations for different objects of the same class. It should therefore be possible to associate *LockBased*, *MultiVersion*, and *Optimistic* to *objects*, not to classes. As a consequence, *LockBased*, *MultiVersion*, and *Optimistic* have to be able to apply the aspects they depend on to their *object*, not to the class.
- *Dynamic Aspects:* In order to support flexible reuse, support for dynamic aspects is required, i.e., it should be possible to apply aspects to and remove aspects from objects at run time. In AspectOptima, for example, in multi-version concurrency control, the *Shared* aspect should be removed from a version of a transactional object when it becomes read-only.
- *Thread-Aware Aspects:* In order to support flexible reuse in multi-threaded applications, it should be possible to activate aspects on a per-thread basis. In AspectOptima, several aspects provide functionality based on the context of the current thread. For instance, *Tracked* only tracks object accesses if the current thread has previously started a tracked zone. *AutoRecoverable* only checkpoints objects if the current thread is within an auto-recoverable zone.

Most of these requirements have been mentioned in the aspect-oriented literature before (the interested reader is referred to the proceedings of the SPLAT (Software Engineering Properties of Languages and Aspect Technologies) [25] and FOAL (Foundations of Aspect-Oriented Languages) [26] workshops). The main contribution here is that AspectOptima requires *all* of the features in order to be implemented in an elegant reusable way.

4.2 AspectJ

We decided to validate the design of AspectOptima and test its effectiveness in evaluating AOP language features by an implementation in *AspectJ* [24]. *AspectJ* is an aspect-oriented extension of the Java [27] programming language. It was conceived by a team of researchers at Xerox Parc and is probably currently the most popular AOP language. The version used for our experiments is version 1.5.2.

In *AspectJ*, crosscutting behavior is encapsulated in a class-like construct called an *aspect*. Similar to a Java class, an aspect can contain both data members and method declarations, but it cannot be explicitly instantiated. Four new concepts introduced in *AspectJ* are relevant to this work: *joinpoints*, *pointcuts, advice*, and *inter-type declarations*.

Joinpoints are well-defined points in the execution of a program. These include method and constructor calls, their executions, field accesses, object and class initializations, and others. Only call and execution joinpoints were essential in our current implementation of AspectOptima.

A *pointcut* is a construct used to designate a set of joinpoints of interest and to expose to the programmer the context in which they occur, such as the current executing object (*this(ObjectIdentifier)*), the target object of a call or execution (*target(ObjectIdentifier)*), and the arguments of the a method call (*args(..)*).

An *advice* defines the actions to be taken at the joinpoint(s) captured by a pointcut. It consists of standard Java code. *AspectJ* supports three types of advice: the *before*, the *after*, and the *around* advice. The *before* advice runs just before the captured joinpoint; the *after* advice runs immediately after the captured joinpoint; the *around* advice surrounds the captured joinpoint and has the ability to augment, bypass, or allow its execution.

Finally, *inter-type declarations* allow an aspect to define methods and fields for other classes.

The following paragraphs of this subsection present techniques and workarounds that can be used in *AspectJ* to achieve some of the requirements presented in Sect. 4.1. It should be noted here that we did not choose *AspectJ* because we expected it to be the ideal language for implementing AspectOptima. To the contrary, there exist many other AOP languages that provide more advanced features and hence are probably more suitable. *CaesarJ* [28], for instance, defines *Aspect Collaboration Interfaces*, which among many other benefits nicely decouple aspect implementations from aspect bindings. Initial experiments with *CaesarJ* however showed that the current compiler is not stable enough to build a complex aspect framework such as AspectOptima.

Separate Aspect Binding and Inter-aspect Configurability. In *AspectJ*, the abstract introduction idiom (also known as indirect introduction) [29, 30] can be used to achieve separate aspect binding and inter-aspect configurability. The abstract introduction idiom allows us to "collect several extrinsic properties from different perspectives within one unit and defers the binding to existing objects" [29]. In order words, the target classes of the static and dynamic crosscutting behavior are unknown until weave-time. This strategy has three participants (see Fig. 5):

- *Introduction container*: a construct used as the target for the inter-type member declarations.
- *Introduction loader*: the aspect that introduces crosscutting behaviors and ancestors to the introduction container.
- *Container connector*: the aspect used for connecting the introduction container to the base application classes.

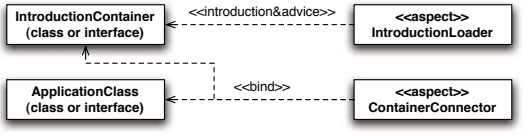


Fig. 5. Abstract Introduction Idiom

The introduction container serves a dual purpose in the context of our implementation. First, it enables the aspects (i.e., both static and dynamic crosscutting behaviors) to be reused in different contexts; second, it helps in identifying the classes to which the crosscutting behaviors of an aspect should be applied.

The introduction container can either be a class or an interface in *AspectJ*. Since multiple inheritance is not supported in Java, our implementation cannot use a class as introduction container: it would prohibit several aspects to be applied to the same application object. Consequently, dummy interfaces are used as the introduction container for each of the aspects. For instance, the interface *IShared* is associated with the aspect *Shared*, *IAutoRecoverable* is associated with *AutoRecoverable*, and so on. Each of the AspectOptima aspects, playing the role of the introduction loader aspect, is then implemented to apply its functionality to all the classes that implement its associated interface (e.g., the *Shared* aspect is applied to all classes that implement the *IShared* interface).

Since all AspectOptima aspects declare dummy interfaces, separate aspect binding can be achieved using the *declare parents* construct of *AspectJ*. The first aspect in Fig. 6 brands the *Account* class as *IShared*; hence, the crosscutting behavior of the *Shared* aspect is applied to all instances of the *Account* class.

Inter-aspect configurability is achieved by having the associated interface of an aspect implement the interfaces of the aspects it depends on. For instance, the *AutoRecoverable* aspect declares *IAutoRecoverable* to implement *IAccessClassified* and *IRecoverable* as illustrated in the second aspect of Fig. 6. Hence, an *AutoRecoverable* object is by default *Recoverable* and *AccessClassified*. This technique to achieve separate aspect binding and inter-aspect configurability

```
public aspect Binding {
  declare parents: Account implements IShared;
}
public aspect AutoRecoverable{
  declare parents: IAutoRecoverable implements IRecoverable, IAccessClassified;
}
```

Fig. 6. Separate Aspect Binding and Inter-aspect Configurability in *AspectJ*

```
public aspect LockBased {
  declare precedence: LockBased,AutoRecoverable,Tracked,Versioned,Shared;
}
```

Fig. 7. Inter-aspect Ordering in *AspectJ*

makes reuse very easy. Application developers do not have to modify their base classes to apply aspects to them.

Inter-Aspect Ordering. Inter-aspect ordering is supported in *AspectJ* by the *declare precedence* construct. Figure 7 illustrates how the *LockBased* aspect specifies its execution order relative to that of the aspects it depends on.

AspectJ precedence declarations are application-wide. It is hence not possible to declare, for instance, two different orderings of the same set of aspects for two different pointcuts. In the current version of AspectOptima, however, such a functionality is not necessary since *MultiVersion* and *Optimistic* depend on the exact same ordering as *LockBased*. However, it is not guaranteed that this would also be the case if the ten base aspects are reused within other applications.

Per-Object Aspects, Dynamic Aspects, and Per-Thread Aspects. *AspectJ* does not support per-object aspects or dynamic weaving. run time enabling and disabling of aspects (i.e., advice within an aspect) can be simulated by introducing a boolean field into each advised object. At each pointcut occurrence, the field is checked to verify that the aspect is actually enabled (see Sect. 4.3 for example code using this technique).

Per-thread aspects can be simulated by using the `ThreadLocal` class provided by the Java standard library. Using `ThreadLocal`, it is possible to associate an object with each thread instance. Within this object, boolean attributes can be stored that can be consulted by the advice of an aspect in order to determine if the aspect is enabled for the currently executing thread or not.

4.3 AspectJ Implementation of AspectOptima

In this section, we present a detailed description of the implementation of some of the AspectOptima aspects in *AspectJ*. Due to space constraints, only the aspects necessary to discuss the encountered *AspectJ* limitations, namely *AccessClassified*, *Copyable*, *Shared*, *Tracked*, and *LockBased*, are presented. The interested reader is referred to [31] for a complete description of the implementation.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Inherited public @interface Read {}

public aspect AccessClassified {
  public enum Kind {READ, WRITE, UPDATE);
  boolean found = false;
  public Kind IAccessClassified.getKind(String methodName)
    throws MethodNotAnnotatedException, MethodNotFoundException{
    for (Method m : this.getClass().getMethods()) {
      if((methodName.trim()).equalsIgnoreCase(m.getName())) {
        found = true;
        if (m.isAnnotationPresent(Read.class)) return READ;
        else if (m.isAnnotationPresent(Write.class)) return WRITE;
        else return UPDATE;
      }
    }
    if (!found) throw new MethodNotFoundException(".."); } }
```

Fig. 8. *AspectJ* implementation of *AccessClassified*

AccessClassified Implementation. In our *AspectJ* environment (based on the `ajc` compiler), it is not possible to statically determine if a method potentially reads, writes, or updates the fields of an object. Therefore, our implementation of *AccessClassified* (see Sect. 3.2) relies on the application developer to tag every method of an object with marker annotations, such as the *Read* annotation defined in the top lines of Fig. 8. The annotations have a run time retention policy (i.e., they are retained by the virtual machine so that they can be read reflectively at run time), can be inherited (i.e., annotations on superclasses are automatically inherited by subclasses), and have to be applied to methods.

The *AccessClassified* implementation aspect shown in Fig. 8 introduces a method (`getKind(String methodName)`) to every *IAccessClassified* object that examines these annotations by reflection at run time and classifies each operation accordingly. Non-annotated methods are treated as modifier operations to guarantee system consistency. For an example of how a developer can classify the operations of a class, see Sect. 4.3.

Another interesting possibility is to determine the access kind automatically at run time by tentatively executing the method and by intercepting all field modifications (see [31] for details). It might also be possible to extend the flexible *AspectJ* compiler `abc` [32] to perform an automatic classification based on static code analysis.

Copyable Implementation. The *Copyable* aspect (Fig. 9) introduces state replacement and cloning functionality to all classes that implement the *`ICopyable`* interface. Java already provides a default `clone()` method for objects that implement the *Cloneable* interface. However, this default method only implements shallow cloning. To provide deep cloning, some additional work is needed. The `replaceState(Object src)` method enables an object to swap its state with

```
public aspect Copyable {
  declare parents: ICopyable implements Cloneable;
  public void ICopyable.replaceState(Object src) {
    try {
      copyFields(this,src);
    } catch (SourceClassNotEqualDestinationClass e) {}
  }
  public Object ICopyable.clone() {
    Object deepCopyOfOriginalObject = super.clone();
    deepCopyOfOriginalObject.replaceState(this);
    return deepCopyOfOriginalObject; } }
```

Fig. 9. *AspectJ* implementation of *Copyable*

that of another object of the same class. The `copyFields(this, src)` helper method performs a field-by-field deep-copy of the state (inherited, declared, and introduced) of the source object to the invoking object (code not shown here for space reasons).

Shared Implementation. The implementation of the *Shared* aspect is presented in Fig. 10. This aspect depends on the method classification provided by the *AccessClassified* aspect to determine the appropriate lock to be acquired for a given operation. The *declare parents* construct used in line 2 illustrates inter-aspect configuration by declaring all *Shared* objects to be *AccessClassified* as well.

Lines 3–5 define a boolean field and two methods for supporting run time disabling and enabling of advice on a per-object basis. The `if(isEnabled(shared))` pointcut modifier on line 9 checks the field before executing the functionality provided by *Shared*.

Lines 6 and 7 allocate a lock for each *Shared* object. The pointcut on line 8 makes sure that all public or protected method executions of a *Shared* object are intercepted. The *around* advice on line 9 obligates every thread executing a method on a *Shared* object to acquire the appropriate lock before proceeding. After the operation is executed, the lock is released again.

Tracked Implementation. Figure 11 presents an implementation of the *Tracked* aspect. It depends on the *AccessClassified* aspect to distinguish between *read*, *write*, and *update* operations, and on the *Named* aspect to avoid tracking different copies of the same transactional object (see line 2). *InheritableThreadLocal,* a class provided by the standard Java API, is used to associate a thread with a zone. The *Zone* class is a simple helper class that maintains three hash tables to keep track of read, written, and updated objects. The code of the *Zone* class is not shown due to space constraints. Tracked zones are requested by executing the aspect method *beginTrackedZone(),* and terminated by executing the aspect method *endTrackedZone()* (see lines 11–14).[4]

[4] For space reasons, the code dealing with joining and leaving, as well as nested zones, has been omitted.

```
 1 public aspect Shared {
 2   declare parents: IShared implements IAccessClassified;
     //Introduce variable and methods for run time disabling/re-enabling of advice
 3   private boolean IShared.Enabled = true;
 4   private boolean IShared.getEnabled() { return Enabled; }
 5   static boolean isEnabled(IShared object) { return object.getEnabled(); }
     //Introduce the variable and method for enforcing synchronization
 6   private Lock IShared.threadLock = new Lock();
 7   private Lock IShared.getSharedLock() { return threadLock; }
 8   pointcut methodExecution(IShared ishared): target(ishared) &&
     (execution(public * IShared+.*(..))||execution(protected * IShared+.*(..)));
 9   Object around(IShared shared) : methodExecution(shared) &&
      if(isEnabled(shared)) {
10     Object obj;
11     Kind accessType = shared.getKind(getMethodName(thisJoinPoint));
12     // Get the appropriate lock
13     if (accessType == READ) shared.getSharedLock().getReadLock();
14     else if (accessType == WRITE) shared.getSharedLock().getWriteLock();
15     else shared.getSharedLock().getUpdateLock();
16     obj = proceed(shared);
     // Release previously acquired lock
...
20     return obj; } }
```

Fig. 10. *AspectJ* Implementation of *Shared*

The pointcut at line 4 makes sure that all public method calls to tracked objects are intercepted. The *before* advice (lines 5–10) only executes if the call is made from within a tracked zone thanks to the *if* pointcut modifier. Line 6 shows how *Tracked* calls `getKind`, a functionality provided by *AccessClassified*. Likewise, line 8 calls `getName`, a functionality provided by *Named*, to obtain the object's identity. If the object has not been associated with the zone yet, the advice records the access according to its category (lines 9 and 10). Finally, lines 15–20 implement operations that provide the set of objects read or modified from within a zone.

LockBased Implementation. The *LockBased* aspect provides support for pessimistic lock-based concurrency control with in-place update (Fig. 12). To accomplish this, it depends on the following aspects: *AccessClassified* (to determine the appropriate transactional lock to acquire for a given transaction), *Shared* (to prevent threads within a transaction from concurrently modifying an object's state), *AutoRecoverable* (to gather undo information in case a transaction aborts), *Tracked* (to keep track of the transactional objects that participate in a transaction), and *Persistent* (to store the state of the object on stable storage when a transaction commits). The inter-aspect configuration is done using the *declare parents* statement of line 2.

The execution order of these aspects is crucial. An unspecified ordering could result in bad performance, deadlock or, in the worst case, even break the ACID

```
1 public aspect Tracked {
2   declare parents: ITracked implements INamed, IAccessClassified;
3   private static InheritableThreadLocal myZone = new InheritableThreadLocal();
4   pointcut methodCall(ITracked track) : target(track) &&
      call(public * ITracked+.*(..));
5   before(ITracked track) : methodCall(track) && if(myZone.get()!= null) {
6     Kind type = track.getKind(getMethodName(thisJoinPoint.toShortString()));
7     Zone z = (Zone)myZone.get();
8     String myName = ((ITracked)thisJoinPoint.getTarget()).getName();
      //Place the target object in the appropriate category
9     if (type == READ && !z.readObjects().containsKey(myName))
        z.addReadObject(thisJoinPoint.getTarget());
10    else if (!z.writeObjects().containsKey(myName))
        z.addWriteObject(thisJoinPoint.getTarget());
    }
11  public static synchronized void beginTrackedZone(){
12    if (myZone.get() == null) myZone.set(new Zone());
    }
13  public static synchronized void endTrackedZone() {
14    myZone.set(null);
    }
15  public static Vector getReadObjects() throws NoZoneFoundException{
16    if (myZone.get() == null) throw new NoZoneFoundException("..");
17    return ((Zone)myZone.get()).getReadObjects();
    }
18  public static Vector getModifiedObjects() throws NoZoneFoundException{
19    if (myZone.get() == null) throw new NoZoneFoundException("..");
20    return ((Zone)myZone.get()).getModifiedObjects(); } }
```

Fig. 11. Implementation of *Tracked*

properties. The desired execution order is *LockBased*, *AutoRecoverable*, *Tracked*, *Versioned*, and *Shared*. *LockBased* first has to acquire the transactional lock and set the update strategy in-place before *AutoRecoverable* executes, the object is then *Tracked*, the operation directed to the main version by *Versioned*, and mutual exclusion to the state of the object ensured by *Shared* as shown in Fig. 2. This ordering is configured using the *declare precedence* statement in line 3.

Lines 4 and 5 allocate an instance of *TransactionalLock* for each lockbased object. The *TransactionalLock* class is a helper class that implements transaction-aware read/write locks. The `acquire` method suspends the calling thread if some other transaction is already holding the lock in a conflicting mode.

The pointcut in line 6 makes sure that all public method calls to a *LockBased* object are intercepted. The *before* advice first queries the current transaction in line 8 (details on transaction life cycle management are out of the scope of this paper). In line 10, the functionality of *AccessClassified* is used to classify the operation that is to be invoked. Line 11 attempts to acquire the transactional lock for the current transaction in the corresponding mode. If successful, line 12

```
1 public aspect LockBased {
2  declare parents: ILockBased implements
     IAccessClassified, IShared, IAutoRecoverable, ITracked, IPersistent;
3  declare precedence: LockBased, AutoRecoverable, Tracked, Versioned, Shared;
4  private TransactionalLock ILockBased.lock = new TransactionalLock();
5  private TransactionalLock ILockBased.getLock() { return lock; }
6  pointcut methodCall(ILockBased lb) : target(lb) &&
     call(public * ILockBased+.*(..));
7  before (ILockBased lb) : methodCall(lb) {
8    Transaction t = getCurrentTransaction();
9    if (t != null) {
10    Kind accessType = lb.getKind(getMethodName(thisJoinPoint.toShortString()));
11     lb.getLock().acquire(t, accessType);
12     lb.setDeferred(false);
   } }
13 before (Transaction t) : call(public void Transaction.commit()) && target(t) {
14   for (ILockBased lb : Tracked.getModifiedObjects()) {
15     lb.saveState();
   } }
16 after (Transaction t) : call(public void Transaction.commit()) && target(t) {
17   for (ILockBased lb : Tracked.getModifiedObjects()) {
18     lb.discardCheckpoint();
19     lb.saveState(); }
20   for (TransactionalLock l : Tracked.getAccessedObjects()) {
21     l.releaseLock(t);
   } }
22 after (Transaction t) : call(public void Transaction.abort()) && target(t) {
23   for (ILockBased lb : Tracked.getModifiedObjects()) {
24     lb.restoreCheckpoint(); }
25   for (TransactionalLock l : Tracked.getAccessedObjects()) {
26     l.releaseLock(t);
   } } }
```

Fig. 12. *AspectJ* Implementation of *LockBased*

sets the update strategy by using functionality provided by *Recoverable* (which is configured to apply to the target object by *AutoRecoverable*).

Unlike *Shared*, *LockBased* follows the two-phase locking protocol, and therefore holds on to the transactional locks until the outcome of the transaction is known. In case of transaction commit, *LockBased* performs the two-phase commit protocol. The first phase is done by the *before* advice on lines 13–15. It obtains all modified objects of the transaction by using the functionality provided by *Tracked* and saves all pre- and post-states to stable storage using the functionality provided by *Persistent*. The second phase is handled by the *after* advice in lines 16–22. It discards the checkpoints of all modified objects using the functionality provided by *Recoverable*, saves their final states to stable storage using the functionality of *Persistent*, and then releases the transactional locks of all accessed objects.

```
1 public class Account implements ILockBased {
2  private float balance;
3  @Read public float getBalance() { return balance; }
4  @Update public void credit(float amount) {balance += amount; }
  }
```

Fig. 13. A Lockbased Account

The *after* advice on lines 22–26 handles transaction abort. It first rolls back all changes made to modified objects using the functionality provided by *Recoverable* and then releases the transactional locks.

Using AspectOptima. Line 1 of Fig. 13 shows how a programmer can declare an application class, in this case the class `Account`, and apply the *LockBased* aspect to it by simply declaring the class to implement *ILockBased*. The `getBalance` and `credit` methods are classified as *read* and *update* operations respectively using the marker annotations of *AccessClassified* in lines 3 and 4.

4.4 Encountered AspectJ Limitations and Possible Improvements

We provide a discussion of the encountered *AspectJ* limitations, possible workaround solutions, and suggestions for improvements to the *AspectJ* language features in this section.

Weak Aspect-to-Class Binding. An object in an *AspectJ* environment could have three types of methods: those inherited from superclasses and superinterfaces, those declared by the class, and those introduced by aspects through direct or indirect introductions. As explained in Sect. 4.2, our implementation achieves aspect reusability, separate aspect binding, and inter-aspect configurability by using the abstract introduction idiom [29, 30]. Extrinsic static crosscutting behaviors are collected in dummy interfaces (via the inter-type member introduction) and these interfaces are later bound to application classes using the *declare parents* construct. For instance, declaring an *Account* class as implementing *ICopyable* introduces two additional public operations: *replaceState(SrcObj)* and *clone()* into every *Account* object.

As described in Sect. 3.2, *Copyable* interferes with *Shared*, in the sense that it should not be possible to copy or clone an object while it is being modified. Assuming that the previous *Account* class also implements *IShared* (such as, for instance, required by the *LockBased* aspect), it seems logical to assume that the call and execution of `Account.replaceState(SrcObj)` will be captured by the pointcuts *call(public * IShared+.*(..))* and *execution(public * IShared+.*(..))* of the *Shared* aspect, since the method *replaceState(SrcObj)* is defined for the *Account* class. This is not the case; the actual call and execution joinpoints are `call(ICopyable.replaceState(..))` and `execution(ICopyable.replaceState(..))`, respectively; i.e, *AspectJ* associates the call and the execution joinpoints of indirectly introduced methods with the *introduction container* not the application class. As a result, *Shared* does not intercept

```
1 placeholder PCopyable {
2   public void clone() {...}
3   public void replaceState(Object o) {...}
  }
4 aspect Copyable {
5   apply PCopyable to Account; }
```

Fig. 14. Proposed "placeholder" Construct

calls to *replaceState(SrcObj)*, which may lead to state inconsistencies if a thread executes a write or update operation while a different thread tries to copy the state of the object. This deficiency is not unique to AspectOptima—any two aspects that interfere and work at the granularity of methods could suffer from the weak aspect-to-class binding problem.

In our case, a possible work-around is to declare the *ICopyable* interface as implementing *IShared*. In this case, the `replaceState` and `clone` method calls are intercepted by *Shared* as desired. An unfortunate side effect though is that *Copyable* is not individually reusable anymore: all *Copyable* objects are now also *Shared*, even if the application is single-threaded. This proposed work-around cannot solve the problem for circularly interfering aspects.

Language Improvement Suggestion: The weak aspect-to-class binding problem could be overcome by adding a new class-like construct to *AspectJ* that we called a *placeholder*. A placeholder can define fields and methods, but these members should not be structurally bound to the placeholder. Its functionality should exclusively be to hold static crosscutting behavior that, at weave time, is bound to the target class it is applied to. A *placeholder* should not be instantiable, should never have a superclass, superinterface, or be part of an inheritance hierarchy.

Figure 14 shows a potential declaration of *PCopyable*, a *placeholder* to be used in the implementation of the *Copyable* aspect. Lines 1–3 define the *placeholder* and the `replaceState` and `clone` methods. Line 5 suggests a new construct for binding the fields and methods of a *placeholder* to the target class, in this case *Account*. As opposed to indirect introduction, this *direct introduction* associates the call and execution joinpoints of fields and methods with the target class. As a result, the use of an interface as an introduction container is no longer necessary. However, in order to use polymorphic calls, an interface declaration for *Copyable* is still needed.

The *placeholder* concept may sound much like mixins [33], but it is fundamentally different. In mixins, the call and execution of a mixin method is delegated to the mixin class, not the target class, and hence the weak aspect-to-class binding problem can occur.

Reflection/Superclass Method Execution Dilemma. The enforcement of the ACID properties of transactional objects occurs at the level of method invocations. To achieve this, the AspectOptima aspects, for instance *Shared*, must rigorously intercept *every* method invocation on a transactional object to perform the appropriate pre- and post-actions before allowing the call to proceed.

AspectJ provides two pointcut designators for intercepting the call and execution of a method: *call(MethodPattern)* and *execution(MethodPattern)*.

The *method call* pointcut can intercept non-reflective calls to *declared* and *inherited* methods of an object, but not reflective calls, i.e., calls using the Java reflection API. For instance, the pointcut *call(public * SavingAccount.*(..))* would intercept the method call `SavingAccount.debit(..)` but not `debit.invoke (SavingAccountObject, ..)`—a conscious design decision made by the *AspectJ* team not to "delve into the Java reflection library to implement call semantics" [34].

The *method execution* pointcut is typically used to address this deficiency. This pointcut can intercept the execution (both reflective and non-reflective) of declared and "overridden-inherited" methods of an object, but unfortunately not the execution (both reflective and non-reflective) of "non-overridden-inherited" methods, because in this case the execution joinpoint occurs in the super class. For instance, the pointcut *execution(public * SavingAccount.*(..))* intercepts both the reflective and non-reflective execution of `SavingAccount.debit(..)`, but not `SavingAccount.getBalance()`, assuming that the `getBalance` method is defined in *Account* and not overridden in the child class *SavingAccount*.

Composing the call and execution pointcuts with an *or* operator is not a feasible solution either, because reflective invocations of `getBalance` can still not be intercepted.

One possible work-around is to require the application programmer to manually override all the inherited methods from a super class in the subclass, in which case the execution pointcut can be used to capture all calls. This solution is however undesirable: the code reuse benefits of inheritance are diminished, methods introduced by aspects cannot be handled without introducing explicit dependencies of the base on the aspect, and there is always the danger that an application programmer forgets to override some of the methods.

Another work-around is to use a pointcut that explicitly names the super class: *target(SavingAccount) && execution(public * Account+.*(..))*. This pointcut intercepts the execution of the methods of an *Account* object when the target is *SavingAccount*. It intercepts both reflective and non-reflective executions of `SavingAccount.getBalance()` and `SavingAccount.debit(..)`. It also correctly excludes the execution of operations on other subclasses of account, e.g. *CheckingAccount*. Unfortunately, this solution is application specific and cannot be reused in a generic context. In order to write the pointcut, the exact superclass and target subclass have to be known.

The only fully generic and reusable solution for the aspect *Shared* would be to write: *target(IShared) && execution(public * *.*(..))*. This pointcut always works, but can result in a significant performance overhead, since a dynamic check has to be performed at *every* public method execution of any class.

Language Improvement Suggestion: We propose the addition of an inheritance-conscious method execution pointcut to *AspectJ*: *superexecution(MethodPattern)*. Given a class with no superclasses, this pointcut behaves exactly as the *execution(MethodPattern)* pointcut (i.e., it intercepts both

declare dependencies: LockBased **requires**
AccessClassified, Shared, AutoRecoverable, Tracked;

Fig. 15. Proposed "declare dependencies" Construct

reflective and non-reflective execution of declared methods). When used on a class with superclasses, it automatically overrides all non-overridden inherited methods, in our case `getBalance()`, within the body of the target class, in our case *SavingAccount*, with dummy methods that simply call the method in the superclass. It then applies the standard *execution(MethodPattern)* pointcut to the class. This ensures that the execution joinpoints of non-overridden inherited methods occur in the target subclass, eliminating the reflection/superclass method execution dilemma problem.

Lack of Support for Explicit Inter-Aspect Configurability. The aspects in AspectOptima exhibit complex aspect dependencies and interferences. *AspectJ* has no construct that enables developers to express inter-aspect configurations. Ideally, an aspect should be able to express the need of functionality offered by other aspects, or adjust its functionality if interfering aspects are applied to the same pointcuts. Also, it should be possible to specify incompatible aspect configurations.

Our *AspectJ* implementation achieves rudimentary inter-aspect configurability by declaring dummy interfaces for each aspect. Aspects express the dependency on other aspects by having their associated interface implement the interfaces of the aspects they depend on using the *declare parents* construct (see, for example, line 2 of Fig. 12). However, this does not guarantee that the aspects are applied to the same joinpoints.

Language Improvement Suggestion: We propose the addition of a new *declare dependencies* construct to *AspectJ*, which would allow inter-aspect configurability to be expressed as proposed in Fig. 15. The desired effect of this line of code is that *AccessClassified*, *Shared*, *AutoRecoverable*, and *Tracked* should be applied to all the joinpoints picked out by *LockBased*. However, general applications might require more fine-grained control over joinpoints in case of complex aspect configurations. Ideally, an aspect should be able to selectively decide to what pointcuts each of the aspects it depends on is to be applied, and on the order in which the advice are to be executed.

Lack of Support for Per-Object Aspects. In systems with many objects, such as in transactional systems, the ability to selectively apply different aspects to different objects of the same class is crucial. For instance, one might want to use pessimistic concurrency control for heavily used *Account* objects, and use optimistic concurrency control for less frequently used instances of the *Account* class. Unfortunately, *AspectJ* does not permit a developer to selectively decide to which instances of a class an aspect should be applied to.

However, the *if(BooleanExpression)* pointcut of *AspectJ* can be used to simulate per-object aspects. An aspect can introduce a field into the target class,

and then test for specific values of that field in the pointcut. For example, an enumeration field *usage* could be introduced into the *Account* class, with possible values of *heavy* and *normal*. The *if* pointcut could inspect the value of the *usage* field to decide if an advice is to be applied to the object or not.

Lack of Support for Run time Disabling and Enabling of Pointcuts. Aspects are statically deployed in *AspectJ*; i.e., the crosscutting behaviors specified in the aspects become effective in the base applications once they are woven together and these crosscutting behaviors cannot be altered at run time. This limitation is encountered, for instance, in multi-version concurrency control (see Sect. 3.3). After an object's state has been committed to history, it does not need to be *AutoRecoverable* and *Shared* anymore, since only *read* transactions are going to access the object's state in the future. To maximize system performance, it should be possible to disable the *AutoRecoverable* and *Shared* aspect for this object.

As shown in the implementation of the *Shared* aspect in Sect. 4.3, the *if(BooleanExpression)* pointcut of *AspectJ* can be used to simulate run time disabling and enabling of aspects. Unfortunately, this only disables the advice associated with a joinpoint. This implies that the target-joinpoint will always be intercepted but the execution of its associated advice is conditional on the value of *BooleanExpression*—resulting in performance loss, since operations of read-only transactions are still unnecessarily intercepted, and a run time check has to be performed on every operation invocation. Reference [35] reports that the *if(BooleanExpression)* pointcut (where *BooleanExpression* is a single static method call) introduces a 22% performance overhead.

Language Improvement Suggestion: Some AOP languages, e.g. JBossAOP [36], already support dynamic weaving of aspects as a whole. One could imagine an even more fine-grained feature that would allow enabling and disabling of pointcuts. For instance, aspects could define static methods `enablePointcut(Pattern)` and `disablePointcut(Pattern)` that would support run time disabling and reenabling of named pointcuts whose name matches *Pattern*. For instance, the call `Shared.aspectOf(obj).disablePointcut(methodExecution)` would disable the method execution interception specified by the *Shared* aspect for the object *obj*, eliminating/reducing the performance overhead.

4.5 Initial Performance Evaluation

We conducted several preliminary performance measurements on our implementation of AspectOptima in order to determine the performance of the aspect-oriented framework, and the performance impact that the lack of support of some of the key language features presented in Sect. 4.1 can have. The measurements are preliminary in the sense that far more measurements would be needed in order to accurately determine the performance impact of aspect-oriented frameworks and language features. In order to compare the performance of different aspect-oriented execution environments, thorough benchmarks should be defined. This is, however, out of the scope of the paper and left for future work.

Table 2. Comparing Object-Oriented and Aspect-Oriented Performance

	OO-read	OO-update	AO-read	AO-update
Time (seconds)	382.897	447.284	1871.734	1945.0231
Overhead (factor)	1	1	4.88	4.35

All our experiments were run on a 3 GHz Intel-based laptop with 512 MB of RAM running Windows XP home edition, Eclipse 3.2.0, Java 1.5, and AspectJ 1.5.2. The measurements were obtained using the Eclipse Test and Performance Tools Platform [37]. All measurements execute operations on a simple bank account class that encapsulates a `balance` field and provides the methods `int getBalance()` and `deposit(int)`.

Aspect-Oriented Implementation vs. Object-Oriented Implementation. This section compares the performance of a purely object-oriented implementation of lock-based concurrency control with our aspect-oriented implementation *LockBased*. To perform the object-oriented measurements, we wrote a wrapper class for the bank account class that overrides `getBalance` and `deposit`, and then calls OPTIMA [9, 17] (the object-oriented version of our framework) to execute the same functionality as *LockBased* and the ten low-level aspects before forwarding the call to the actual account.

The performance measurements are given in Table 2. We performed 50,000 `getBalance` (read) operations and 50,000 `deposit` (update) operations. The overall slowdown of the aspect-oriented implementation is around 1490 s for both read and update operations, which represents 30 ms per operation.

The fact that the aspect-oriented implementation is slower is not surprising. Each of the low-level aspects is individually reusable and does not know about the specific context in which it is used. This independence makes it impossible to share run time information among aspects. For instance, *LockBased* has to query the access kind of the method to be called from *AccessClassified*. But so does *AutoRecoverable*, *Tracked*, and *Shared* (see Fig. 2). The object-oriented implementation however can optimize: it calls *AccessClassified* only once, and then passes the access kind as a parameter to the different components implementing 2-phase locking, recovery, and mutual exclusion.

This is of course not a problem of aspect-orientation as such, but rather a problem of separation of concerns in general. Since each aspect should be individually reusable, it cannot depend on other aspects to classify the operation. It is foreseeable, however, that this slowdown in the future will become less significant thanks to advances in compiler and weaving technology. For instance, *LockBased*, *AutoRecoverable*, *Tracked* and *Shared* all apply to the same joinpoint. An advanced weaver, such as found in the `abc` [32] compiler or the *Steamloom* environment [38], might be able to detect this situation and perform context-dependent optimizations. To make this possible, the compiler would have to detect that the result returned by `getKind(String methodName)` of *AccessClassified* is constant for a given method name. It always returns the same meta-information.

Table 3. Performance Overhead due to Lack of Dynamic Aspects

	not shared	shared & not enabled	shared (no `if`)	shared & enabled
Time (seconds)	0.586430	8.755212	54.023821	63.328594
Overhead factor	1	15	92	108

Performance Impact of Simulating Per-Object Aspects. The need for per-object aspects and dynamic aspects, i.e., runtime disabling and re-enabling of aspects, is motivated by the multi-version concurrency control example. Once an object's state is committed, it is inserted into the history and is subsequently only ever accessed by read-only transactions. Hence, the functionality provided by the *Shared* aspect is not needed anymore, since no transaction will ever modify that particular version of the object's state in the future. In *AspectJ* it is not possible to disable the pointcut defined in the *Shared* aspect at runtime. An *if(BooleanExpression)* pointcut modifier has to be used to simulate the disabling as shown in lines 3–5 and 9 of Fig. 10. Since the *AspectJ* rules forbid the use of non-static function calls within the boolean expression, an additional static version of the `getEnabled()` method that simply forwards the call to the target object had to be created.

To measure the performance overhead incurred, we performed three experiments, in which the read-only operation `getBalance` was called 1,000,000 times. The results of the experiment are presented in Table 3.

The first column shows the time spent inside `getBalance` for a standard bank account object. The third column shows the time spent inside `getBalance` when *Shared* has been applied to the bank account object (but in this case without an `if` pointcut modifier in the pointcut). This includes the call to *AccessClassified* and the acquisition of the read lock. Obviously, the time spent in the method is considerably bigger—in our case by a factor of 92. The overhead of the `if` pointcut modifier is apparent in the second and the last column. They show the time it takes to check if the shared aspect is enabled for a particular bank account object. Our experiments show a slowdown of 8.2 s (a factor of 15!) when shared is disabled, and a slowdown of 9.3 s when it is enabled.

An aspect-oriented environment that supports dynamic aspects can therefore achieve significantly better performance. Of course, the actual activation/deactivation of aspects at runtime might also be costly. However, very often activation and deactivation are rare events, and their overhead can be safely ignored. In the case of multi-version concurrency control, the *Shared* aspect is deactivated once and for all when the object's state is inserted into the history of states.

Performance Impact of Writing Reusable Pointcuts. The last experiment we conducted aimed at evaluating the performance loss incurred in *AspectJ* due to having to work around the reflection/super class execution dilemma. In Sect. 4.4, we described that with a targeted *call* pointcut we cannot handle reflective calls, whereas with a targeted *execution* pointcut we cannot handle

Table 4. Comparing Application-Specific and Reusable Pointcuts

	targeted read	targeted update	generic read	generic update
Time (seconds)	40.210723	29.900585	65.503984	52.767678
Overhead (factor)	1	1	1.63	1.76

executions of methods defined in the super class. The only way to achieve full functionality and reusability is to write a generic pointcut that intercepts *all* public method executions occurring in the application and dynamically check for the specific target at runtime.

To evaluate the performance loss we again ran 1,000,000 `getBalance` and `deposit` operations on a shared bank account object, once using the targeted, application-specific execution pointcut *target(SavingAccount) && execution(public * Account+.*(..)*, and once with the generic, reusable execution pointcut *target(IShared) && execution(public * *.*(..))*. The results are presented in Table 4.

The table shows that *read* operations are slower than *update* operations. This results from the fact that acquiring a read lock takes in general more time than acquiring a write lock.

The results also show that the slowdown resulting from a generic pointcut is not too significant: less than a factor of 2. This result must however be interpreted carefully. The performance loss measured here is the loss that is incurred due to the generic pointcut when calling a *Shared* object. But the generic pointcut will slow down *every public method execution* in the system, regardless of whether the object is shared or not, and therefore results in huge runtime overhead for an application with many calls to methods of non-shared objects.

5 Related Work

The ideas and techniques investigated in this paper intersect with a broad spectrum of research projects on transactional systems and aspects, reusable aspect-oriented frameworks, and aspect dependencies and interactions. We present the most relevant related work in this section.

5.1 Aspects Implementing General Application Concerns

Aspects for Concurrent Programming. Cunha et al. [3] investigated techniques for implementing reusable aspects for high-level concurrency mechanisms in *AspectJ*. The authors illustrated how abstract pointcut interfaces and annotations can be used to implement one-way calls, synchronization barriers, reader/writer locks, and schedulers. The performance overhead and reusability of an object-oriented implementation of these mechanisms was compared to their aspect-oriented implementation. They concluded that the *AspectJ* implementation is more reusable and pluggable, but incurs a noticeable performance overhead. However, *AspectJ* was found to have a limitation in acquiring local

joinpoint information in concrete aspects: when a superaspect defines an abstract pointcut, the subaspects cannot change the pointcut's signature.

Similar to our work, Cunha et al. used annotations to denote methods that require special processing at run time. However, their work did not address the issue of aspect dependencies and interactions that may occur when these concurrency mechanisms are applied to a common method. In addition, their technique used in supporting aspect reusability is different and requires additional code. In their case, developers must provide concrete pointcuts and advice for each of the abstract pointcuts, which can be error-prone if done incorrectly. Conversely, the *declare parents* construct used by our AspectOptima implementation to bind the aspects to application classes is safe: the correct pointcuts are hardcoded in the aspects.

Persistence Aspects. Rashid et al. [5, 39] have worked extensively on techniques which apply AOP concepts to database systems. In [5], the authors explored three issues in the context of AOP and data persistence: the possibility of using AOP techniques in aspectizing persistence, the reusability of persistence aspects, and whether persistence aspects could be developed independently of an application. Using a relational database application as an example, they demonstrated incrementally how reusable aspects for database connections, data storage and updates, data retrieval, and data deletion can be implemented in *AspectJ*. It was concluded that persistence can indeed be aspectized in a reusable way, but can only be partially developed independently of an application since operations such as data retrieval and deletion must be explicitly considered.

Rashid et al. achieved reusability by requiring all classes whose instances are to be made persistent extend a common base class. Since Java does not support multiple inheritance, classes that already extend other classes cannot be made persistent without some degree of code restructuring. In contrast, reuse in AspectOptima is achieved through marker interfaces—aspects are applied to classes that implement their associated interfaces—eliminating concerns about multiple inheritance support. In addition, our *Persistence* aspect is built upon reusable aspects that can be useful in non-transactional contexts too, whereas their implementation uses database specific code that cannot be reused in other contexts. On the other hand, our *Persistence* aspect currently does not support databases.

Aspects for Distributed Applications. In [40], the authors proposed JAC, an AOP-based middleware for building distributed Java applications. JAC separates aspect binding and crosscutting code into two different modules, respectively aspect components and dynamic wrappers, facilitating aspect reuse. Support for run time aspect deployment is achieved through load-time transformations, whereas our aspects must all be weaved in at compile-time and then selectively enabled or disabled. To achieve aspect distribution, JAC offers a container mechanism that hosts both application objects and aspect component instances and makes them remotely accessible using either CORBA or RMI.

Similar to our transaction aspect, developers can reuse the aspects that come with the JAC framework (e.g., tracing, persistence, authentication, session, load-balancing, to name a few) easily within their application. Configuration is achieved through a separate configuration file. JAC differs from AspectOptima because the functionality that JAC provides is not achieved by composing individually reusable base aspects, and hence aspect dependencies and interferences also are not addressed in a general and reusable way. The JAC documentation does not explicitly mention aspect dependencies and interferences, but it is safe to assume that they are handled internally for every possible aspect combination.

5.2 Aspects and Transactions

The work of Fabry et al. [41, 42] applies AOP concepts to advanced transaction models, e.g. nested and long running transactions. The authors argued that the high complexity and inadequate separation of concerns in these models impedes their use by application programmers. In order to encourage the use of these models, they proposed a domain-specific aspect language called KALA for modularizing the concepts of advanced transaction models into aspects. KALA is based on the ACTA formalism [43].

The main goal of our work is to define an aspect framework with many individually reusable aspects that can be combined in several ways according to the application developers needs. AspectOptima is not meant to be used in a high-performance transactional application, but rather serves as a real-world aspect framework for experimenting with intricate aspect dependencies and interactions. While KALA aims at synthesizing new transaction models and at providing an elegant interface for defining transaction boundaries to an application programmer, our framework simply aims at implementing several concurrency control and recovery strategies by combining individually reusable aspects in different ways.

5.3 Reusable Aspect Design Frameworks

Our ten aspects can be combined in different ways to implement different concurrency control and recovery strategies. A few design frameworks promise this kind of customizable composability, and we present some of them later. This version of our design did not follow any particular methodology, so it would be interesting future work to compare it with similar designs obtained through some of the following frameworks.

FODA. Feature-Oriented Domain Analysis (FODA) [44] is a domain analysis method for product line development, i.e., a family of systems in a domain, rather than a single system. Domain products, representing the common functionality and architecture of applications in a domain, are produced from domain analysis. Specific applications in the domain may be developed as refinements of the domain products.

As part of the process, FODA prescribes the creation of a feature model. Features are defined as attributes, properties, functions, capabilities, or services

of a system that directly affect end-users. The feature interaction model allows the developer to specify dependencies among features, such as *specialization*, *optional, requires*, *mutually exclusive with*. When building a concrete application, the developer has to specify which features the final application should contain.

FODA is a domain analysis method, and hence very different from aspect frameworks such as AspectOptima. An interesting experiment would be to apply FODA to the transaction domain and compare the *user-oriented* features identified in the FODA feature model with the AspectOptima aspects.

Framed Aspects. In [45] the authors propose *framed aspects*, an approach that uses AOP to modularize crosscutting and tangled concerns and frame technology [46] to allow aspect parameterization, configuration, and customization. In framed aspects, the identification of features (here called feature aspects), and detection of dependencies and interferences, is performed following the high-level feature interaction approach promoted by FODA. Once this is done, the features are modularized within *framed aspects*, together with their dependencies.

Framed aspects are made up of three distinct modules: the framed aspect code (normal and parameterized aspect code), composition rules (aspect dependencies, acceptable and incompatible aspect configurations), and specifications (user-specific customization). These modules are composed to generate customized aspect code using a frame processor. Framed aspects achieve separate aspect bindings and aspect dependencies through parameterization and composition rules respectively. Composition rules can also be used for specifying acceptable and incompatible aspect configurations. The above mentioned constructs enable framed aspects to be reused in contexts other than that for which they were implemented.

It would be very interesting to attempt an implementation of AspectOptima using framed aspects in order to investigate how well this approach supports the language features presented in Sect. 4.1.

Ahead. Reference [47] describes a generic mechanism to generate a group of artifacts (code, documentation, makefiles, uml diagrams...) which, together, describe a program implementing a given subset of the available features. To do so, they first choose a labelled-tree representation for each artifact, and they interpret the resulting structure as a hierarchy of objects containing other objects. They then label the containment relationships according to the features they implement, extracting similarly purposed relationships into independently injectable mixins, and grouping the mixins into "layers" representing the features.

This mixin-based strategy is strikingly similar to CaesarJ's collaboration composition mechanism [28], and quite different from AspectJ's strategy. Attempting an implementation with this kind of language should be especially insightful.

6 Conclusions

Designing and implementing the ACID properties of transactional objects is a simple, but non-trivial, real-world example to which aspect-oriented techniques can be applied. The first part of the paper presents AspectOptima, a

language-independent, aspect-oriented framework that ensures the ACID properties for transactional objects. The framework consists of ten base aspects at the lowest level, each one providing a well-defined reusable functionality. The base aspects are simple, yet have complex dependencies among each other. We demonstrated how the base aspects can be configured and composed in different ways to implement different concurrency control and recovery strategies. This composition is delicate: some aspects conflict with each other or require a specific invocation ordering, others have to be reconfigured dynamically at run time.

All of the above, and the fact that AspectOptima has not been invented to promote a particular aspect-oriented system, makes it an ideal challenge case study for the aspect-oriented community. In particular, we believe that it can be used to evaluate:

- *Aspect-Oriented Software Development Processes:* We performed our decomposition into aspects based on our previous implementation experience. How does an AOSD process perform when applied to this case study? Is the resulting decomposition equally simple, modular, and reusable? Can the process identify aspect dependencies and conflicts?
- *Aspect-Oriented Modeling Notations:* Can an AOM notation capture the complex structural and behavioral dependencies of the aspects in this case study? Are the resulting models easy to understand? Are the models reusable? Can aspect dependencies and conflicts be expressed?
- *Aspect-Oriented Validation and Verification:* Can AO formal methods and AO testing techniques be used to individually validate and verify each aspect in a stand-alone way? Can these methods detect conflicts between aspects?
- *AOP Implementations:* How do different AOP implementations compare with respect to performance and memory footprint? How do they compare to standard OO implementations? Is run time weaving to implement dynamic AOP faster than static weaving with run time checks?
- *Aspect-Oriented Language Features:* What are the features that an AO programming language must provide in order to implement this case study? What features can promote good software engineering properties such as encapsulation, modularization, testability, maintainability, and reusability? What features are required to support aspect dependencies and resolution of conflicts in a reusable way?

In order to give an idea on how AspectOptima can be used to evaluate the expressiveness of aspect-oriented languages, we presented in the second part of this paper an implementation of AspectOptima in *AspectJ*. We identified six key language features that an aspect-oriented language must provide in order to implement AspectOptima in a satisfactory way: separate aspect binding, inter-aspect configurability, inter-aspect ordering, per-object aspects, dynamic aspects, and per-thread aspects. We then showed parts of our implementation to demonstrate that *AspectJ* provides sufficient, but certainly not ideal or elegant, support for implementing reusable aspect frameworks and dealing with mutually dependent and interfering aspects. We discussed the encountered

language limitations, suggested possible language improvements where appropriate, and presented some preliminary measurements that highlight the performance impact of certain language features.

7 Future Work

We have demonstrated how AspectOptima can be used to evaluate the language features of *AspectJ*, and intend to do the same for other AOP languages in the immediate future. We intend to define different benchmarks that can be used to compare the performance of these AOP environments, and run these benchmarks on our implementations to obtain reference measurements. We also plan to run the benchmarks on the object-oriented implementation of Optima [9, 17] and compare the results.

We are also working on extending AspectOptima. First of all, concurrency control and recovery can be further enhanced when more information about the semantics of operations is available. We intend to define a *SemanticClassified* aspect that defines forward and backward commutativity for all operations of an object, and maps every operation to a corresponding inverse operation. Such semantic information opens the door to semantic-based concurrency control [11] and logical recovery based on intention lists.

The work described in this paper has focused on the identification and implementation of aspects that crosscut objects. However, the *Versioned*, *Tracked*, and *AutoRecoverable* aspects share a common need for a well-defined *region of computation* (zone/view) that threads can be associated with, and during which certain actions (such as object accesses) are to be monitored. This is a crosscutting concern of *threads of computation*, rather than a crosscutting concern of objects. We have already started to extend AspectOptima beyond object-centered aspects, implementing transaction life-cycle management with aspects. Initial results can be found in [48].

Acknowledgments

This research has been partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors would also like to thank Jean-Sebastien Légaré and Isaac Yuen for their work on the implementation of the AspectOptima prototype, and the anonymous reviewers.

References

[1] Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H.: Discussing Aspects of AOP. Communications of the ACM 44, 33–38 (2001)

[2] Soares, S., Laureano, E., Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. In: Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 174–190. ACM Press, New York (2002)

[3] Cunha, C.A., Sobral, J.L., Monteiro, M.P.: Reusable Aspect-Oriented Implementations of Concurrency Control Patterns and Mechanisms. In: Proceedings of the 5th International Conference on Aspect-Oriented Software Development - AOSD 2006, pp. 134–145. ACM Press, New York (2006)
[4] Kienzle, J., Guerraoui, R.: AOP - Does It Make Sense? The Case of Concurrency and Failures. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 37–61. Springer, Heidelberg (2002)
[5] Rashid, A., Chitchyan, R.: Persistence as an Aspect. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development - AOSD 2003, pp. 120–129. ACM Press, New York (2003)
[6] Kienzle, J., Gélineau, S.: AO Challenge: Implementing the ACID Properties for Transactional Objects. In: Proceedings of the 5th International Conference on Aspect-Oriented Software Development - AOSD 2006, March 20-24, 2006, pp. 202–213. ACM Press, New York (2006)
[7] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, San Mateo (1993)
[8] Kienzle, J., Romanovsky, A., Strohmeier, A.: Open Multithreaded Transactions: Keeping Threads and Exceptions under Control. In: Proceedings of the 6th International Worshop on Object-Oriented Real-Time Dependable Systems, Roma, Italy, 2001, pp. 209–217. IEEE Computer Society Press, Los Alamitos (2001)
[9] Kienzle, J.: Open Multithreaded Transactions — A Transaction Model for Concurrent Object-Oriented Programming. Kluwer Academic Publishers, Dordrecht (2003)
[10] Papadimitriou, C.: The Serializability of Concurrent Database Updates. Journal of the ACM 26, 631–653 (1979)
[11] Ramamritham, K., Chrysanthis, P.K.: Advances in concurrency control and transaction processing, Los Alamitos, California (1997)
[12] Kung, H.T., Robinson, J.T.: On Optimistic Methods for Concurrency Control. ACM Transactions on Database Systems 6, 213–226 (1981)
[13] Bernstein, P.A., Goodman, N.: Concurrency Control in Distributed Database Systems. ACM Computing Surveys 13, 185–221 (1981)
[14] Papadimitriou, C.H., Kanellakis, P.C.: On Concurrency Control by Multiple Versions. ACM Transactions on Database Systems 9, 89–99 (1984)
[15] Agrawal, D., Sengupta, S.: Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control. In: Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon, pp. 408–417. ACM Press, New York (1989)
[16] Lampson, B., Sturgis, H.: Crash Recovery in a Distributed Data Storage System. Technical report, XEROX Research Center, Palo Alto (1979)
[17] Kienzle, J., Jiménez-Peris, R., Romanovsky, A., Patiño-Martinez, M.: Transaction Support for Ada. In: Strohmeier, A., Craeynest, D. (eds.) Ada-Europe 2001. LNCS, vol. 2043, pp. 290–304. Springer, Heidelberg (2001)
[18] Kienzle, J., Romanovsky, A.: A framework based on design patterns for providing persistence in object-oriented programming languages. IEEE Proceedings of Software Engineering 149, 77–85 (2002)
[19] Riehle, D., Siberski, W., Bäumer, D., Megert, D., Züllighoven, H.: Serializer. In: Pattern Languages of Program Design, vol. 3, pp. 293–312. Addison-Wesley, Reading (1998)

[20] Krasner, G., Pope, S.: A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Journal of Object-Oriented Programming 1, 26–49 (1988)
[21] Kienzle, J., Strohmeier, A.: Shared Recoverable Objects. In: Harbour, M.G., la de Puente, J.A. (eds.) Ada-Europe 1999. LNCS, vol. 1622, pp. 397–411. Springer, Heidelberg (1999)
[22] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., Morrison, R.: An Approach to Persistent Programming. Computer Journal 26, 360–365 (1983)
[23] Eswaran, K.P., Gray, J., Lorie, R.A., Traiger, I.L.: The Notion of Consistency and Predicate Locks in a Database System. Communications of the ACM 19, 624–633 (1976)
[24] Kiczales, G., Hilsdale, E., Hugunin, J., Kersen, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–357. Springer, Heidelberg (2001)
[25] Workshop on Software Engineering Properties of Languages and Aspect Technologies – SPLAT (2003 - 2007)
[26] Workshop on Foundations of Aspect-Oriented Languages – FOAL (2002 - 2007)
[27] Gosling, J., Joy, B., Steele, G.L.: The Java Language Specification. The Java Series. Addison Wesley, Reading (1996)
[28] Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An overview of caesarJ. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)
[29] Hanenberg, S., Costanza, P.: Connecting Aspects in AspectJ: Strategies vs. Patterns. In: First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (2002)
[30] Hanenberg, S., Unland, R.: Parametric Introductions. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development - AOSD 2003, pp. 80–89. ACM Press, New York (2003)
[31] Duala-Ekoko, E.: Evaluating the Expressivity of AspectJ in Implementing a Reusable Framework for the ACID Properties of Transactional Objects - Master Thesis, School of Computer Science, McGill University (2006)
[32] Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: an Extensible AspectJ Compiler. In: AOSD 2005: Proceedings of the 4th international conference on Aspect-oriented software development, pp. 87–98. ACM Press, New York (2005)
[33] Schmidmeier, A., Hanenberg, S., Unland, R.: Known Concepts Implemented in AspectJ. In: 3rd Workshop on Aspect-Oriented Software Development (AOSD-GI) of the SIG Object-Oriented Software Development, German Informatics Society (2003)
[34] Xerox Corporation: Frequently Asked Questions about AspectJ (2006), http://www.eclipse.org/aspectj/doc/released/faq.html
[35] Hilsdale, E., Hugunin, J.: Advice Weaving in AspectJ. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development - AOSD 2004, pp. 26–35. ACM Press, New York (2004)
[36] Burke, B., Chau, A., Fleury, M., Brock, A., Godwin, A., Gliebe, H.: JBoss Aspect-Oriented Programming (2004)
[37] The Eclipse Project: Eclipse Test and Performance Tools Platform (2007), http://www.eclipse.org/tptp/

[38] Bockisch, C., Arnold, M., Dinkelaker, T., Mezini, M.: Adapting Virtual Machine Techniques for Seamless Aspect Support. In: ACM Sigplan International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 109–124 (2006)
[39] Rashid, A.: Aspect-Oriented Database Systems. Springer, Heidelberg (2004)
[40] Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., Martelli, L.: JAC: an Aspect-Based Distributed Dynamic Framework. Software Practice and Experience 34, 1119–1148 (2004)
[41] Fabry, J., Cleenewerck, T.: Aspect-Oriented Domain Specific Languages for Advanced Transaction Management. In: International Conference on Enterprise Information Systems 2005 (ICEIS 2005) proceedings, pp. 428–432. Springer, Heidelberg (2005)
[42] Fabry, J., D'Hondt, T.: KALA: Kernel Aspect Language for Advanced Transactions. In: SAC 2006: Proceedings of the ACM Symposium on Applied Computing, pp. 1615–1620. ACM Press, New York (2006)
[43] Chrysanthis, P.K., Ramamritham, K.: Synthesis of Extended Transaction Models using ACTA. ACM Transactions on Database Systems 19, 450–491 (1994)
[44] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
[45] Loughran, N., Rashid, A.: Framed Aspects: Supporting Variability and Configurability for AOP. In: International Conference on Software Reuse (ICSR-8), pp. 127–140. Springer, Berlin (2004)
[46] Bassett, P.: Framing Software Reuse: Lessons from the Real World. Prentice-Hall, Inc., Upper Saddle River (1997)
[47] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering 30, 355–371 (2004)
[48] Bölükbaşi, G.: Aspectual Decomposition of Transactions. Master's thesis, School of Computer Science, McGill University, Montreal, Canada (2007)

Analysis of Aspect-Oriented Model Weaving

Katharina Mehner[1,⋆], Mattia Monga[2,⋆⋆], and Gabriele Taentzer[3,⋆⋆]

[1] Software Engineering Group, Technical University of Berlin, Germany
mehner@cs.tu-berlin.de
[2] Dip. di Informatica e Comunicazione, Università degli Studi di Milano, Italy
mattia.monga@unimi.it
[3] Dep. of Computer Science and Mathematics, Philipps-Universität Marburg, Germany
taentzer@mathematik.uni-marburg.de

Abstract. Aspect-oriented concepts are currently exploited to model systems from the beginning of their development. Aspects capture potentially crosscutting concerns and make it easier to formulate desirable properties and to understand analysis results than in a tangled system. However, the complexity of interactions among different aspectualized entities may reduce the benefit of aspect-oriented separation of crosscutting concerns. Some interactions may be intended or may be emerging behavior, while others are the source of unexpected inconsistencies. It is therefore desirable to detect inconsistencies as early as possible, preferably at the modeling level.

We propose an approach for analyzing interactions and potential inconsistencies at the level of requirements modeling. We use a variant of UML to model requirements in a use case driven approach. Activities that are used to refine use cases are the joinpoints to compose crosscutting concerns. The activities and their composition are formalized using the theory of graph transformation systems, which provides analysis support for detecting potential conflicts and dependencies between rule-based transformations. This theory is used to effectively reason about potential interactions and inconsistencies caused by aspect-oriented composition. The analysis is performed with the graph transformation tool AGG in order to get a better understanding of the potential behavior of the composed system. In addition, the activity control flow of the aspect/base specification and the composition operators are taken into account to identify the relevant interactions.

1 Motivation

Aspect-oriented programming promises to provide better separation and integration of crosscutting concerns than plain object-oriented programming. Aspect-oriented concepts are currently introduced in all phases of the software

⋆ This author has been supported by the German Federal Ministry for Education and Research (BMBF) under the grant 01ISC04 (Project TOPPrax).
⋆⋆ These authors have been supported by the EU Research Training Network SegraVis.

A. Rashid and H. Ossher (Eds.): Transactions on AOSD V, LNCS 5490, pp. 235–263, 2009.

development life cycle with the aim of reducing complexity and enhancing maintainability already early on.

On the requirements level, crosscutting concerns, i.e., concerns that affect many other requirements, cannot be cleanly modularized using object-oriented and view-point-based techniques. Several approaches have been proposed to identify crosscutting concerns already at the requirements level and to provide means to modularize, represent, and compose them using aspect-oriented techniques, e.g., for use case-driven modeling in [1–4].

A key challenge is to analyze the interaction and consistency of crosscutting concerns with each other and with affected requirements. It is in particular the quantifying nature [5] of aspect-oriented composition that makes the detection of interactions and inconsistencies difficult. When composing aspect-oriented and object-oriented models, there are two sources of *interactions*, and thus of potential inconsistencies.

1. Intended or unintended overlap of concepts in these models can be the source of inconsistencies. Depending on the composition, these inconsistencies become effective or are avoided.
2. Aspect-oriented composition specifies where and when an aspect is applied and how the control flow is augmented or changed. It can be used to solve some of the above-mentioned inconsistencies, e.g., by replacing object-oriented behavior consistently with aspect-oriented behavior. However, it might create inconsistencies by accidentally duplicating or suppressing behavior.

Requirement engineers should identify aspect interactions and potential inconsistencies since they could compromise the feasibility of the system. However, it is worth noting that not all identified interactions are necessarily inconsistencies. Some of them may be intended or emerging collaborations. Until now, approaches to analyzing the aspectual composition of requirements have been informal [3, 4, 6]. Formal approaches for detecting inconsistencies have been proposed only for the level of aspect-oriented programming, e.g., model checking [7], static analysis [8], and slicing [9, 10].

Techniques proposed for the programming level cannot be used for requirements because they rely on the operational specification of the complete behavior as given by the code, while requirements abstract from these details. On the requirements level, a commonly used, but often informal, technique is to describe behavior with pre- and post-conditions, e.g., using intentionally defined states or attributes of a domain entity model. This technique is, e.g., used for defining UML use cases, activities, or methods. In order to allow a more rigorous analysis of behavior, this approach has to be formalized and also extended to aspect-oriented units of behavior.

In this paper, we investigate the use of an existing model analysis technique based on *graph transformations* [11] for analyzing interactions and inconsistencies of an aspect-oriented composition of object and aspect models. The rule-based paradigm of graph transformation can be used as a formal model for behavior specifications with pre- and post-conditions. The theory provides

results for detecting potential interactions and inconsistencies among behavioral specifications using a technique called critical pair analysis.

We illustrate our approach with a *use case-driven* modeling approach using UML [12] use case, activity, and class diagrams. We specify aspect-oriented compositions for use cases using their refining activities as joinpoints. The composition is formally defined as model weaving on the level of activity diagrams using graph transformation techniques. Activities are rigorously defined with pre- and postconditions using a variant of UML and subsequently analyzed for conflicts and dependencies with the tool AGG [13], an environment for specifying, analyzing, simulating, and executing graph transformation systems. Since the graph transformation system is not aware of aspects, the results have to be interpreted according to the aspect-oriented composition specification and according to the control flow between the activities of the aspect/base use case.

This paper takes the ideas presented in [14] a step further. Here, we present a formalization of the aspect-oriented composition that was only informally introduced in our previous work. We also give more details on how to use the conflicts and dependencies computed with AGG in order to formally reason about interactions on the level of aspect-oriented composition. In our previous work, this was introduced only by an example.

The paper is organized as follows. Section 2 describes the formally enhanced use case-driven approach using an example and introduces the notion of conflicting and depending behavioral interactions. In Sect. 3, we introduce graph transformations and their analysis facilities. The aspect-oriented composition as model weaving using graph transformations is formalized in Sect. 4. Section 5 explains how to use critical pair analysis to formally analyze interactions in aspect-oriented models. In Sect. 6, we apply the analysis to the example and interpret results with respect to aspect-oriented composition. Related work is discussed in Sect. 7. Section 8 contains our conclusion and outlook.

2 Aspect-Oriented Requirements Modeling

Several authors have proposed extending a use case-driven requirements modeling approach by aspects [1, 6, 15, 16]. Aspects can represent functional and non-functional crosscutting requirements. In [17], functional aspects are identified at the level of use case relationships. The *joinpoints*, i.e., the points of aspect-oriented composition, are activities or groups of activities, as in [6]. We present a subset of these techniques in order to demonstrate how such approaches can be enhanced by a formalization and a formal analysis.

2.1 A Use Case-Driven Approach

Central to our approach is the notion of use case diagrams which serves as an overview on functionality. In addition, a use case is at least specified by a trigger, an actor, a pre-, a post-condition, main scenario(s), and exceptional scenario(s). Scenarios can be specified with UML activity diagrams. Here, we only present

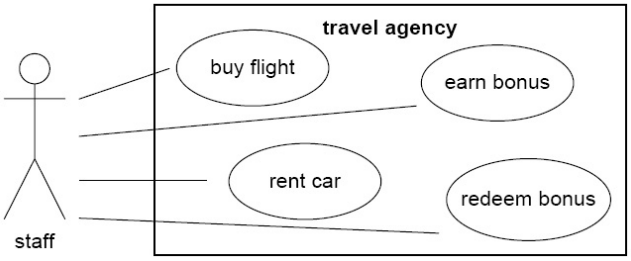


Fig. 1. Use Cases in the Travel Agency Example

scenarios since they are the ones which will be formalized. In addition, the domain model class diagram plays an important role, because we refer to it in the formalization.

We illustrate the approach using a model of a travel agency offering flights reservation, car rental, and using a bonus scheme.

Use Cases. For purchasing travel items, the system offers the use cases "buy flight" and "rent car" (Fig. 1). The use cases "earn bonus" and "redeem bonus" offer a bonus program. A staff member is involved as an actor in all use cases but this does not imply that the actor always triggers the use case.

Domain Model Class Diagram. The class diagram specifies the structure of the domain (Fig. 2). A *Customer* may book and pay for a *Travel* item, either a *Flight* or a *Rental*, sold by an *Agency*. Each travel item can be booked at most

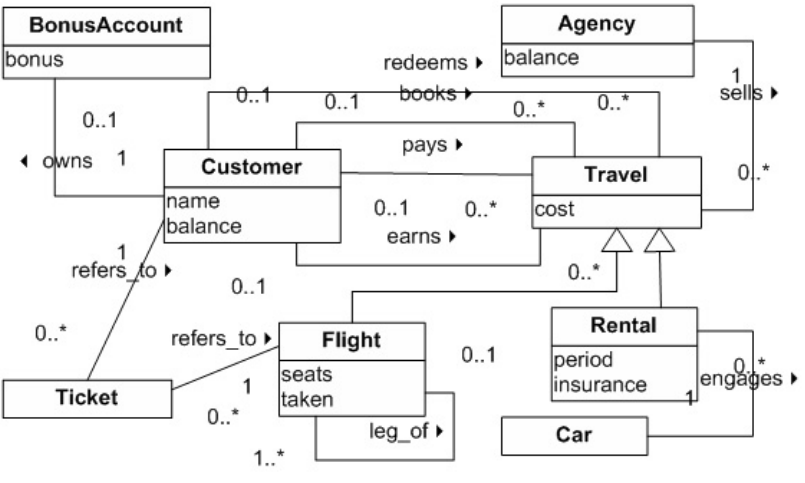


Fig. 2. Domain Model Class Diagram

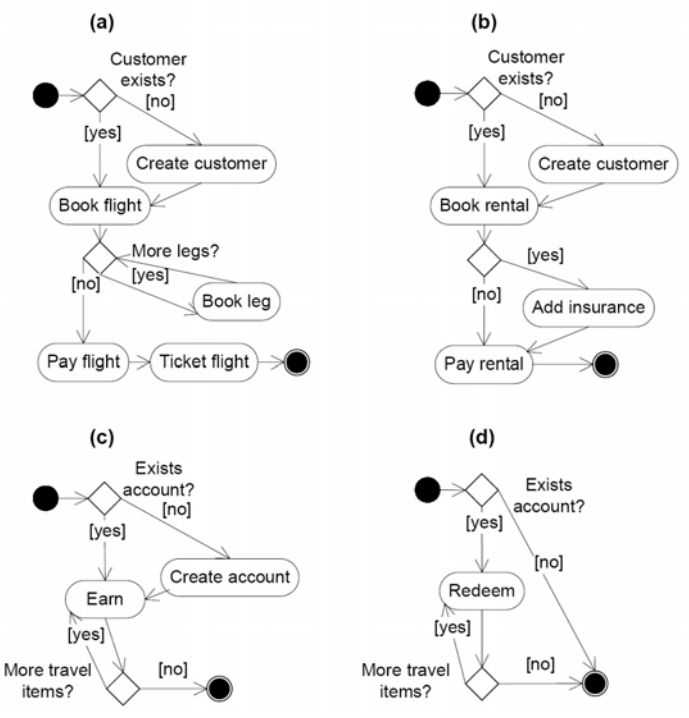


Fig. 3. Activity Diagrams

once. A *Flight* is composed of one or more legs, denoted by "leg_of". A *Ticket* "refers_to" a *Customer* and a *Flight*. A *Rental* "engages" one *Car*. A *Car* can be engaged in different *Rentals*. A *Customer* who "owns" a *BonusAccount* may earn and redeem bonus for *Travel* items.

Activities. The steps of the main scenario of each use case are described using activity diagrams. The use case "buy flight" is refined in Fig. 3a. After conditionally creating a customer, the flight and all its legs are booked. Then the flight is paid for and a ticket issued. The use case "rent car" is specified analogously in Fig. 3b. Bonus use cases are independent of the kind of travel. To earn a bonus, a bonus account must exist. A bonus is earned for all travel items (Fig. 3c). For redeeming a travel item, one uses the bonus (Fig. 3d).

Pre- and Post-conditions. The domain model can be more tightly integrated with activities by specifying the pre- and post-conditions of each activity using prototypical instances. An object diagram, i.e., the structural part of a UML collaboration diagram, is naturally complemented by a diagrammatic description of such a pre- or post-conditions. This has also been advocated by object-oriented methods like Fusion [18] or Catalysis [19]. The pre- and post-conditions serve to specify the intended behavior of an activity. The semantics of the pre-condition is

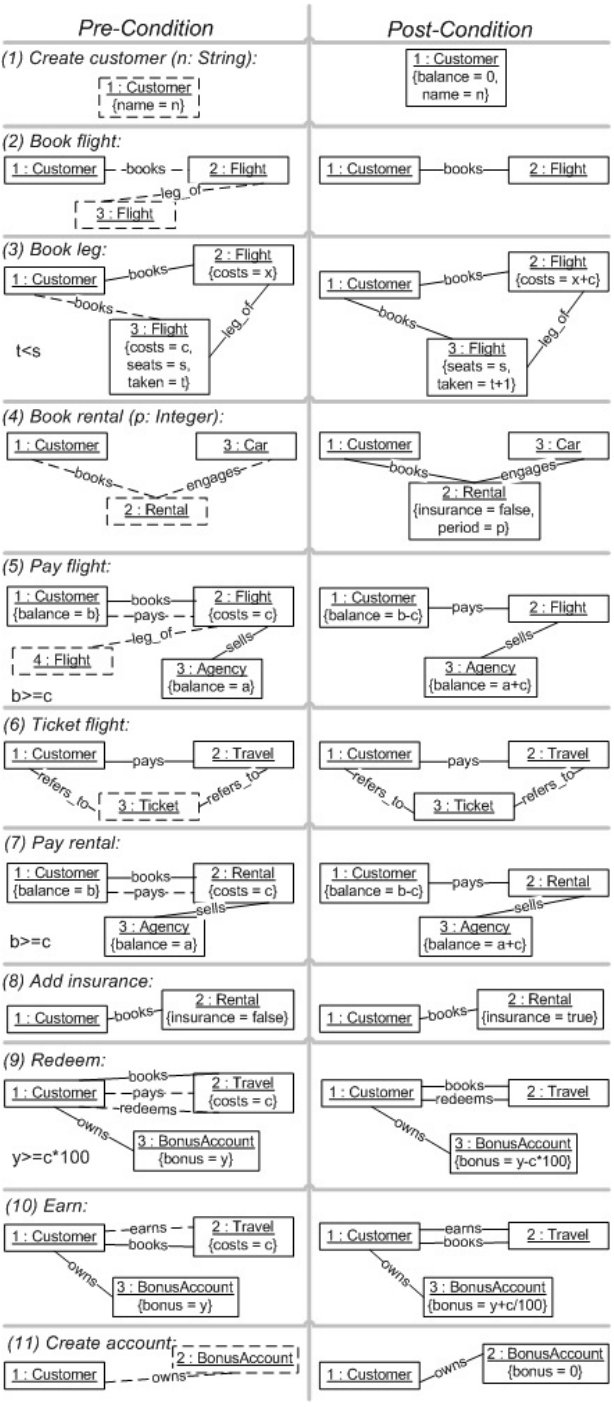


Fig. 4. Activity Pre- and Post-Conditions

that when the pre-condition is violated, the activity cannot be executed. Instead, an exceptional state is reached. In the following, we present the technical details.

The pre- and post-conditions specify arbitrary but fixed instances. Instance identity is given by a natural number. A post-condition can refer to the *same instance* as the pre-condition by referring to the same number. Also, for links, numbers are required. Here, we have omitted them to make it easier to read the conditions. Links between two instances that have the same identity in the pre- and post-condition also have the same identity. *Attributes* can be matched with values. An attribute to be changed in a post-condition has to be instantiated in the pre-condition. *Creation* is specified by introducing a new instance or link in a post-condition. *Deletion* is specified by omitting an instance or a link present in a pre-condition in the corresponding post-condition. A pair of pre- and post-conditions can be parameterized and the parameter can be used in the pre- and in the post-condition (Fig. 4(1)). Pre-conditions can include *negative* conditions, which are often used to specify that a created element does not yet exist. Negative links and instances are drawn using a dashed (out)line using a notation coming originally from graph transformation. If there are several negative elements, each of them is prohibited, i.e., several negative elements have an AND-semantics. (OR-semantics is also possible, but not discussed here.)

Figure 4 gives the pre- and post-conditions of all activities. In Fig. 4(1), the pre-condition checks that a customer named "n" does not exist. The post-condition ensures that this customer is created. In Fig. 4(2), neither a "books" link exists, nor is the flight a "leg of" another flight. A link "books" is inserted. In Fig. 4(3), attributes are matched with value parameters that are used to calculate the post-condition values. In the pre-condition, a constrained on the values "t<s" is used. The other pre- and post-conditions are constructed in a similar way. When specifying these conditions, the formal analysis of the tool AGG and its simulation capabilities (Sect. 3) already helped in identifying unintended imprecisions.

2.2 Aspect-Oriented Composition

Until now, we have left the specification of aspect-oriented relationships between use cases open. The notion of *aspect-oriented composition* is analogous to AspectJ [20]:

- An *advice* is modeled by a use case and subsequently specified using activity diagrams and pre- and post-conditions.
- The *pointcuts*, i.e., the matching specifications, refer to activities. Each activity can thus be a *joinpoint*. A pointcut can refer explicitly to an activity or a list of activity. In order to support crosscutting, activities can be tagged with an additional classification label that can be used in a pointcut.
- The modifiers `before`, `after` and `replace` indicate that the advice use case is executed before, after, or instead of each activity matched.
- A *condition* specifies when the aspect activity diagram can be executed. If the condition is not fulfilled, the aspect activity is not executed and the base behavior is left unchanged.

We have chosen not to use lexical patterns (as used in previous work [14]) in activity matching as names of activities are often given in a natural language and as they often contain short phrases and not only a single word. This impedes the lexical matching approach. The classification can be implemented, for instance, using UML tagged values or even stereotypes.

It is also conceivable to have more complex joinpoints. In [6], in addition, groups of activities, e.g., an entire use case, can be matched. Here, we do not consider the *proceed*-mechanism [20] inside the `replace` advice, which allows the behavior associated with a joinpoint (here it is an activity) to be used inside the advice. This would require to introduce a special place holder activity that is used in the aspect activity diagram to mark the place holder activity identified by a pointcut.

The condition of the aspect-oriented composition does specify the condition under which an aspect activity diagram can be executed. If the condition is not fulfilled, the aspect activity is not executed. In case of `before` or `after`, the ordinary base activity is always executed. In case of a conditional `replace`, the base activity is executed instead. Thus, this condition has an effect on the control flow of an activity diagram.

In the example of the travel agency use cases (Fig. 1), crosscutting behavior is exhibited by the use case "earn bonus". It augments the use cases "buy flight" and "rent car". Thus, the activity diagram specifying "earn bonus" is composed with the other activity diagrams. It should take place after booking is completed, i.e., before the following activity "Pay flight" or "Pay rental".

In order to support the crosscutting idea, the two pay activities are grouped in a classification labeled "Pay", to this end a new tagged value or a new stereotype could be added to the UML models for the flight booking. The pointcut matches activities that belong to the classification "Pay" (Table 1) using the modifier `before`. "Redeem bonus" should take place instead of "pay flight" or "pay rental"(Table 1). The composition will be formalized in Sect. 4.

Table 1. Aspect-Oriented Composition

Use Case	Modifier	Pointcut (Activity/Classification)	Condition
Earn bonus	`before`	–/Pay	Customer requests bonus
Redeem bonus	`replace`	–/Pay	Customer wants to redeem

2.3 Interactions in Aspect-Oriented Composition

During aspect-oriented modeling, one needs to understand the effects of an aspect model on the model of the rest of the system, i.e., other aspect models and object models, but also how the aspect model is affected by them. The specified aspect-oriented composition should be feasible and should not violate other behavioral constraints. This issue has been further analyzed by Katz [21], who distinguishes the following desirable properties of aspect-oriented composition:

- Specified properties of the existing system are preserved (apart from replaced behavior).
- The aspect adds desired new properties.
- Different aspect behaviors do not interfere.

These desirable properties are affected by the two sources of interactions identified in Sect. 1, interactions directly between behavior and interactions established through the aspect-oriented composition. We can identify these interactions based on the activities specified with pre- and post-conditions.

An activity A *impedes* an activity B if B cannot take place after activity A because the pre-conditions of B are violated by the post-conditions of A. An activity A *enables* an activity B if A produces something needed by the activity B or deletes something forbidden by B. These interactions might arise between an activity from the object model and between an activity from the aspect model in both directions or between activities of different aspect models.

In the following, we analyze all the interactions which become effective with regard to aspect-oriented composition. Some of them may be tolerable or even needed as a function of the application domain; however, our formal analysis helps in making them explicit.

Through the aspect-oriented composition two control flows are merged, and activities from different models become direct or indirect successors or predecessors of each other or replace each other. All the interactions should be taken into account in order to determine if the merge is successful, i.e., if the additional behavior is enabled and not prevented by conflicts and if it does not change the existing behavior.

We illustrate the typical scenario with the use case "redeem bonus" (Fig. 1). The aspect-oriented composition (Table 1) specifies that its activities (Fig. 3) can replace an activity classified as "Pay" (Fig. 3). To check that the composition can work, we have to compare the pre- and post-conditions of the activities involved in order to establish potential interactions between activities. In the example, one would try to find out whether "Redeem" can occur after "Book flight" and "Book rental". This is possible because it depends on them.

Manually identifying all the interactions from pre- and post-conditions is inefficient and error-prone. In the next section (Sect. 3), we describe how the detection of interactions can be automated using existing technologies and tool support. We therefore introduce the basics of graph transformation theory which are used to formalize pre- and post-conditions of behavioral models.

3 Graph Transformation

The UML variant presented in Sect. 2 is a modeling approach for requirements which can be precisely defined by the theory of graph transformation. While class structures are formalized by type graphs, pre- and post- conditions of activities are mapped to graph rules. The formalization functions as the necessary basis for analyzing interactions in aspect-oriented composition precisely. The calculus of

graph transformation has a sound background dating back to the early seventies: the interested reader is referred to seminal work in this area [11]. In this paper, we present the theoretical background in an almost informal way.

3.1 Attributed Typed Graph

Graphs can be used as an abstract representation of diagrams. A graph is defined by the sets of its vertices and edges as well as two functions, source and target, that map edges on vertices. According to this definition, more than one edge can exist between two given vertices. When formalizing object-oriented modeling, graphs occur at two levels: the type level (defined based on class diagrams) and the instance level (given by all valid object diagrams). This idea is described by the concept of *typed graphs*, where a fixed *type graph* TG serves as an abstract representation of the class diagram. Moreover, both vertices and edges may be decorated by a number of *attributes*, i.e., names with a value and type. As in object-oriented modeling, types can be structured by an inheritance relation. Instances of the type graph are object graphs equipped with a structure-preserving mapping to the type graph, i.e., a mapping that preserves the source and target functions for edges. A class diagram can thus be represented by a type graph, plus a set of constraints over this type graph, expressing multiplicities and perhaps further constraints.

In our running example, the type graph (Fig. 5) represents the *domain model* of the system, equivalent to the UML class diagram in Fig. 2. However, the inheritance relationship was rendered by *flattening* all the associations of Travel to Flight and Rental. This is necessary because all the edges of a graph should have the same semantics (a relationship between two nodes) to be used consistently during the analysis. Figure 6 shows an instance graph consistent with the type graph.

3.2 Attributed Typed Graph Transformations

Basically, a *graph transformation* is a rule-based modification of a graph G into a graph H. Rules are expressed by two graphs (L, R), where L is the left-hand side of the rule and R is the right-hand side, and a mapping between objects in L and R. The left-hand side L represents the pre-conditions of the rule, while the right-hand side R describes the post-conditions. $L \cap R$ (the graph part that is not changed) and the union $L \cup R$ should form a graph again, i.e., they must be compatible with source, target and type settings, in order to apply the rule. Graph $L \setminus (L \cap R)$ defines the part that is to be deleted, and graph $R \setminus (L \cap R)$ defines the part to be created.

Figure 4(3) shows pre- and post-conditions of the activity "Book leg", which can be interpreted as a graph rule. The numbers indicate the mapping between left- and right-hand sides. The attribute conditions are interpreted as an instantiation of variables on the left-hand side, and attribute assignment on the right-hand side.

A *graph transformation step* $G \overset{r,m}{\Longrightarrow} H$ between two instance graphs G, H is defined by first finding a match m of the left-hand side L of a rule r in the current

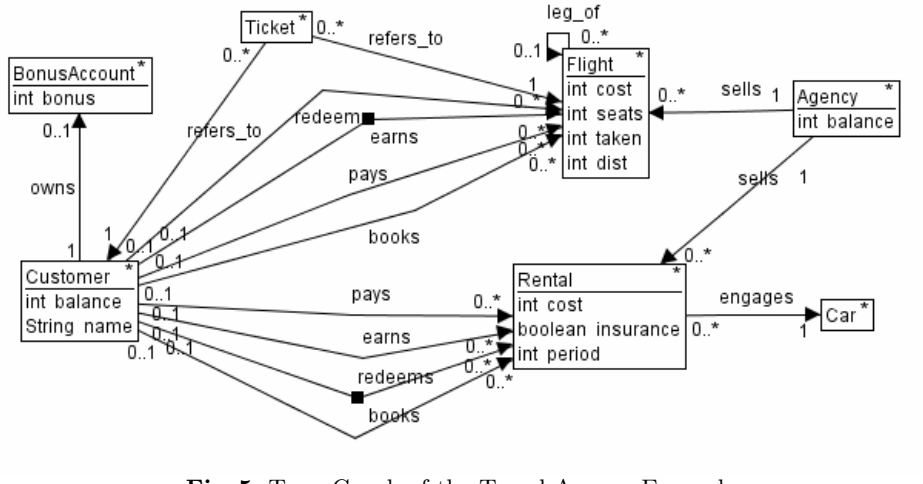


Fig. 5. Type Graph of the Travel Agency Example

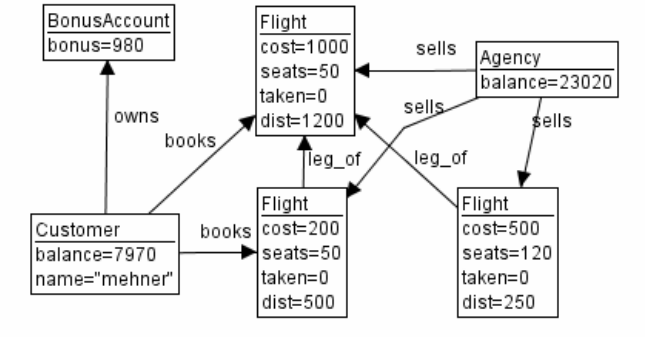


Fig. 6. Instance Graph

instance graph G such that m is structure-preserving and type-compatible. If a vertex embedded into the context is to be deleted, edges that would not have a source or target vertex after rule application might occur: these are called *dangling edges*. There are mainly two ways to handle this problem: either the rule is not applied at match m or it is applied and all dangling edges are also deleted. In the sequel we adopt the former strategy.

Performing a graph transformation step that applies rule r at match m, the resulting graph H is constructed in two passes: (1) build $D := G \setminus m(L \setminus (L \cap R))$, i.e., delete all graph items that are to be deleted; (2) $H := D \cup (R \setminus (L \cap R))$, i.e., create all graph items that are to be created. A *graph transformation*, more precisely a graph transformation sequence, consists of zero or more graph transformation steps.

The applicability of a rule can be further restricted by additional application conditions. The left-hand side of a rule formulates some kind of positive

condition. In certain cases, *negative application conditions* (NACs), which are pre-conditions prohibiting certain graph parts, are also needed. If several NACs are formulated for one rule, each of them has to be fulfilled (AND-semantics). See, e.g., rule "Pay flight" in Fig. 4(5), which has two NACs, one forbidding the flight to be paid for to be a leg of another flight and one checking if the payment has already been recorded.

As an example of a graph transformation step, we consider again the rule "Book leg" in Fig. 4(3) and the host graph in Fig. 6. There are different ways of matching the rule to the host graph, depending on which flight leg is used. Choosing the left flight leg, the NAC (indicated by dashed lines in Fig. 4) is not fulfilled. Thus, the rule can only be applied to the right flight leg. The result is the host graph in Fig. 6 with an additional "books"-edge to the right flight leg.

A set of graph rules, together with a type graph, is called a *graph transformation system* (GTS). A GTS may show two kinds of non-determinism: (1) For each rule several matches may exist. (2) Several rules might be applicable. There are techniques to restrict both kinds of choices. The choice of matches can be restricted by either letting the user specify part of the match using, e.g., input parameters, or by explicitly defining a control flow over rule application.

The tool environment AGG (Attributed Graph Grammar System) [13] can be used to specify graph transformation systems and analyze their rules. Moreover, the rules can be tested by applying them to possible instance graphs.

3.3 Conflict and Dependency Analysis

One of the main static analysis facilities for GTSs is the check for conflicts and dependencies between rules and transformations, both supported in AGG. This conflict and dependency analysis is based on critical pair analysis. The main result of this analysis is the Critical Pair Lemma which states that local confluence of all graph transformations can be gained if all critical pairs (presenting conflicts in minimal context) are local confluent. The critical pair analysis is part of the comprehensive theory on algebraic graph transformation, summarized in [11]. First results on critical pair analysis for graph transformation, especially hypergraph rewriting, were given in [22].

The dependency analysis and a critical pair visualization for algebraic graph transformation have been investigated recently, partly motivated by the aspect-oriented composition analysis presented in this paper. The conflict analysis has been applied, e.g., to identify conflicts in functional requirements in [23]. In this paper, we argue that the existing theoretical results for graph transformation can be used for analyzing potential conflicts and dependencies in aspect-oriented modeling.

As discussed in the previous section, graph transformation systems can show certain kinds of non-determinism. Considering the case where two graph transformations can be applied to the same host graph, the result might be the same, regardless of the application order. Otherwise, if one of two alternative transformations is not independent of the second, the first will disable the second. In this case, the two rules are in *conflict.* Conversely, two transformations

are said to be *parallel independent* if they modify different parts of the host graph. Instead, *sequential independence* guarantees that the order of application in a transformation sequence does not matter, i.e., performing the first transformation does not disable the second.

Analyzing the conflicts and dependencies of concrete graph transformations can be compared to testing a program at run time. The analysis would have greater value if conflicts and dependencies could be detected during compile time by a static analysis. A promising approach in this direction is the analysis of potentially conflicting situations by *critical pairs*. A critical pair is a pair of transformation steps $G \overset{p_1,m_1}{\Longrightarrow} H_1$, $G \overset{p_2,m_2}{\Longrightarrow} H_2$ that are in conflict in a minimal context, identified through matches m_1 and m_2. Host graph G is constructed based on overlapping L_1 and L_2, the left-hand sides of rules p_1 and p_2. The set of critical pairs represents precisely all potential conflicts, i.e., there exists a critical pair as above if and only if p_1 may disable p_2 or, conversely, p_2 may disable p_1. Conflicts can be of the following types:

delete/use: The application of p_1 deletes a graph object that is used by the match of p_2.
produce/forbid: The application of p_1 produces a graph structure that a NAC of p_2 forbids.
change/use: The application of p_1 changes an attribute value of a graph object that is used by the match of p_2.

Critical pair analysis can also be used to find all potential dependencies among rules. In fact, a rule p_1 is a dependency for p_2 if and only if there exists a critical pair between p_1^{-1} (obtained by exchanging the left and right-hand sides of p_1) and p_2. Consequently, the following dependencies are possible:

produce/use: The application of p_1 produces a graph object that is needed by the match of p_2.
delete/forbid: The application of p_1 deletes a graph objects that a NAC of p_2 forbids.
change/use: The application of p_1 changes an attribute of a graph object that is used by the match of p_2.

We use critical pair analysis to detect potential conflicts and dependencies among crosscutting specifications. Before describing our analysis approach in detail in Sect. 5, we will formalize the weaving of crosscutting specifications in Sect. 4.

4 Model Weaving

In this section, we formalize the aspect-oriented composition informally introduced in Sect. 2. We use again the graph transformation approach that was presented in the previous section. We give the semantics of the composition of an aspect use case with a base use case by constructing an activity diagram

that results from composing the base activity diagram with the aspect activity diagram according to the pointcut specification.

Since its coining, the term aspect-oriented programming has always been a synonym for implementing aspects using *weaving*, i.e., for a transformation of the source code which inserts the aspect code in all places specified by a pointcut. Here, we apply the same concept to the activity diagram of the aspect-oriented use case. This diagram is woven into the activity diagram of the base use case following the specification of the pointcut that was introduced in Sect. 2.

In our modeling approach, a pointcut specifies all activities that have to be augmented by an aspect, either by a list of activities or by annotations. Any activity can be a joinpoint. The pointcut specifies a certain mode, i.e., `before`, `after`, or `replace`. This determines where the activity diagram is woven in with respect to an activity specified in the pointcut. In addition, a condition controls the dynamics of the activation of aspect activities.

Thus, we can formalize the weaving by a transformation that inserts the conditional branching for a selected activity. For specifying this kind of transformation, graph transformations are suitable. Graph transformations as introduced in the last section are intended to work on the abstract syntax. The abstract syntax of any visual language can be defined in terms of nodes and edges. Here, we consider activity diagrams that consist of

- activity nodes, start nodes, ends nodes,
- fork and join nodes,
- decision and merge nodes,
- flow edges, and the special flow edges labeled with decision (yes, no).

For these kinds of activity diagrams it is feasible to specify the transformations on the abstract syntax. To specify the transformation, a corresponding type graph and rules based on this type graph have to be defined. Both will be presented in the following.

4.1 Type Graph

In the type graph a super type `Node` subsumes all nodes in the activity diagram (Fig. 7). This allows rules that apply to all kind of nodes to be defined for the super type. `Fork`, `join`, `start`, `stop`, and `merge` nodes do not carry any additional information. A node for an `activity` has an attribute for the name and for the annotation. The latter was introduced in Sect. 2 to support crosscutting by means of annotating a classification. A node for a `decision` contains the predicate for the condition.

An edge is formalized as a `Transition` node that is linked by source and target edges to that nodes the edge connects. A transition node can exist between any two nodes with the exceptions that start and end nodes have no incoming rsp. outgoing edges. These constraints can be formalized as graph constraints in the graph transformation formalization. All other constraints address other well-formedness issues and are not further discussed here. A transition node

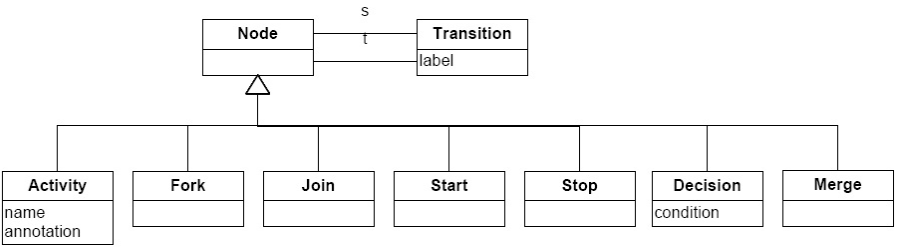


Fig. 7. Type graph for activity diagrams

connected via its source to a decision node is labeled by *Yes* or *No*. All other nodes have empty labels.

4.2 Transformation Rules

We specify a weaving rule for each of the modifiers that can be used in an aspect-oriented composition. Graph transformation rules as introduced in Sect. 3 can transform graph structures with a pre-defined number of elements only. If we want to adapt an activity identified by a pointcut, this activity can have an arbitrary number of incoming edges. In order to capture an arbitrary number of elements in a rule, we introduce the concept of *amalgamated* rules.

Rule Amalgamation. Here, we can only give a short overview on amalgamation of rules. We will focus on the way how an amalgamation is defined. We will not give details on its formalization in the graph transformation approach, which are given in [24–26]. An amalgamated rule can have multi-nodes in addition to ordinary nodes. While each ordinary node has to be matched exactly once, multi-nodes are matched as often as possible. Thereby, a rule can match an arbitrary number of nodes of the same type linked by the same edge type to the context. An example will be given in the following.

Composition Using the `before`, `after` and `replace` Modifiers. In the following we show how to specify a rule for each modifier. Each rule is parameterized with the classification of the activities that can be matched and the condition. Rules for pointcuts matching concrete activities are analogous. The aspect activity diagram is not given as a parameter but can be specified as a **partial** match when applying this rule. A partial match can be specified for a rule application to constrain rsp. control the match of the rule application. The reason is that in the underlying formalism parameters cannot be graphs.

The `before` composition is specified in Fig. 8. The left-hand side of the rule specifies the match of the activity that is augmented (on the left) and the match for an activity diagram (on the right), which can be given as a partial match through its start and end node. The start node has one outgoing transition, while the end node can have multiple incoming transitions. The node after the start node can be an activity, a fork, or decision. The activity specified in the pointcut can have an arbitrary number of incoming edges, here transition nodes,

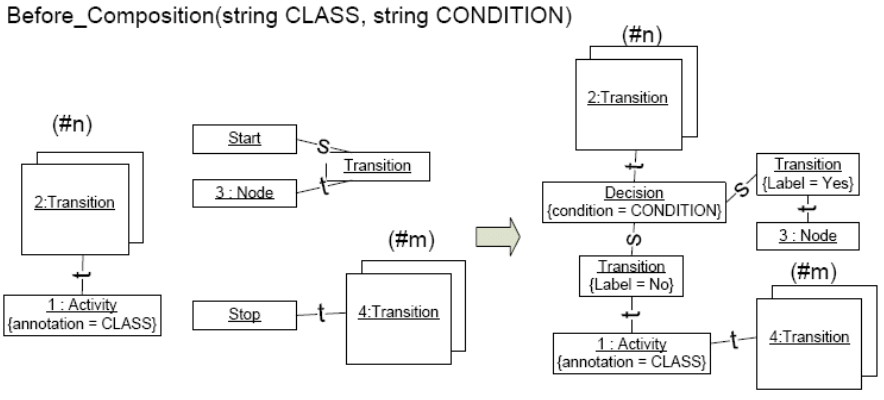


Fig. 8. Before_composition

that have to be redirected. The corresponding multi-node is linked to the newly created decision node. The label of this decision node is given the parameter *CONDITION*. The label $\sharp n$ indicates that the number of objects matched is preserved.

The activity diagram to be woven with the matched activity has start and end nodes that have to be deleted. The first node linked with the start node is preserved. However, its incoming edge, depicted as transition node, has to be replaced by a transition node that carries the label *Yes*. The end node may have an arbitrary number of transition nodes for incoming edges that also have to be redirected. This is captured by a multi-object whose number of instances is preserved. Its outgoing links are attached to the activity matched. Note that we have omitted the numbers of the edges. The rule for `after` is constructed analogously and is not shown here.

The rule for `replace` composition is given in Fig. 9. Also, the replacement is conditional. This transformation has to redirect the edges going into the stop node to the node that is the successor of the replaced activity. Again, this situation is captured by rule amalgamation. Also, note that the link of the matched activity to its m successor nodes is preserved.

4.3 Weaving Example

We apply the rule `Before_Composition` to weave the activity diagram for the use case "earn bonus" with the activity diagram for use case "pay rental" (Fig. 3b,c). The weaving follows the composition specified in Table 1.

The rule requires two parameters and a partial match for the aspect activity diagram which is woven into the base activity diagram. The parameters are given in the composition specification in Table 1. The annotation string "Pay" is passed to the parameter *CLASS*. The condition string "Customer requests bonus" is passed to the parameter *CONDITION*. When applying the parameterized left-hand side of the rule to the abstract syntax of the activity diagram for "pay

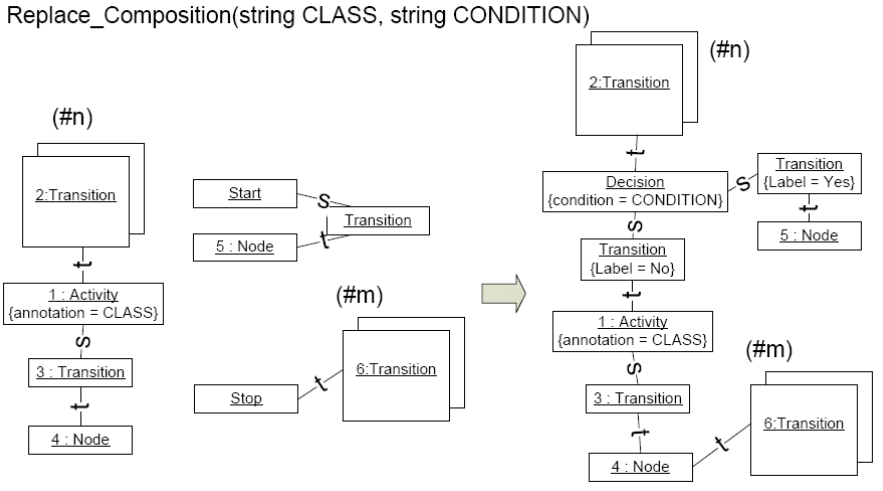


Fig. 9. Replace_composition

rental", the match found is given on the left in Fig. 10. The activity that is matched through the annotation "Pay" hat the name "Pay rental". The multi-node for incoming transitions matches two incoming transitions, one with a label "no" that originates in a decision node and one with no label that originates in an activity node. That is the minimal context needed for a precise match. Note that applying the rule would also result in a match of the activity "Pay flight" of the activity diagram for use case "pay flight". This effect is not shown here.

The rule also needs a partial match for the activity diagram that is to be woven. The partial match is given by the start node context and the end note context of the activity diagram to be woven. The partial match for the activity diagram for use case "earn bonus" is given on the right in Fig. 10. It consists of the start node and of the one transition leaving the start node and of the successor node. Here, the successor is a decision node with the condition "Exists account?". The other part of the partial match is given through the end node and its potentially multiple incoming transitions. Here, the end node has one incoming transition labeled "no" because it originates in a decision node.

The rule preserves all nodes from the match in the base activity diagram and it deletes the start and end node from the activity diagram to be woven. It adds a decision node and puts together both diagrams by adding required transition nodes and edges. The final result is shown on the level of concrete syntax in Fig. 11. The activity diagram for earning bonus is inserted before the the activity "Pay rental" using a decision node with the label "Customer requests bonus". The newly created elements are highlighted.

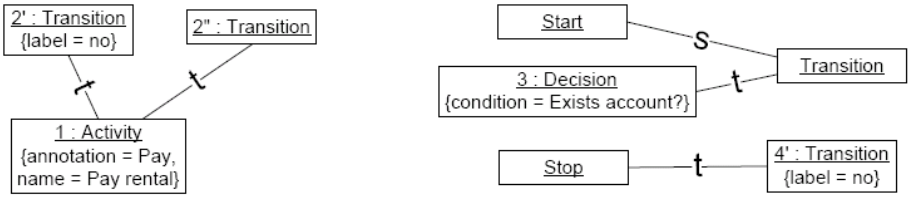


Fig. 10. Example weaving match

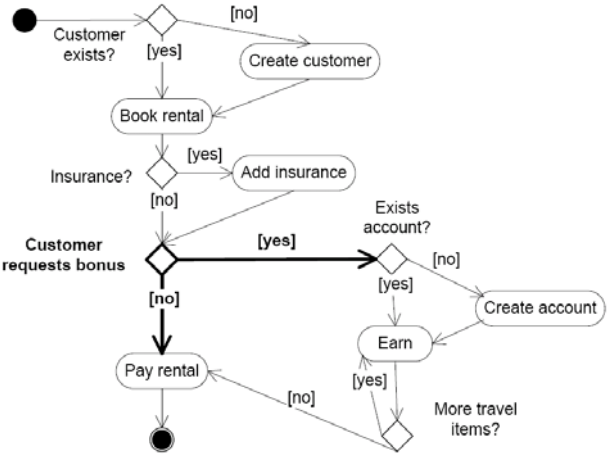


Fig. 11. Example weaving result

5 Applying Critical Pair Analysis

Critical pair analysis can be exploited to discover dependencies among activities. In particular, it is possible to check if a given activity B cannot take place immediately after an activity A. In this case we say that A *impedes* B. We assume (see Section 2.1) that activities are specified by expressing their pre- and post-conditions as graphs: an activity is modeled by a production rule that has the pre-condition as the left-hand side and the post-condition as the right-hand side. Let A and B be two activities specified respectively by the rules p_A and p_B. If the rule p_A is in conflict with the rule p_B, the application of p_A (corresponding to the execution of A) makes p_B inapplicable, i.e., at least one of the constraints imposed by the left-hand side of p_B cannot be satisfied.

A delete/use conflict is shown, e.g., in Fig. 12. Applying "Pay_flight" to the host graph shown at the bottom of the figure, rule "Redeem_flight" becomes non-applicable, because "Pay_flight" deletes the "books"-edge which is needed for the application of "Redeem_flight".

Another conflict occurs if a Customer has booked both a Flight and a Rental and wants to redeem loyalty points from her/his BonusAccount for both. The

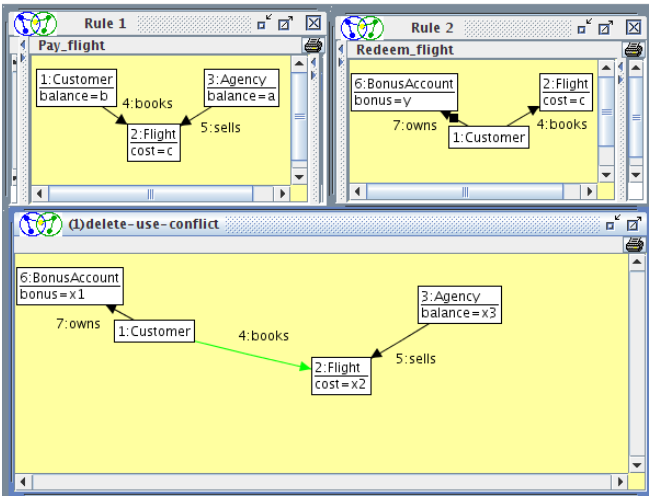


Fig. 12. Critical Pair "Pay_flight", "Redeem_flight" in AGG

rules "Redeem_flight" and "Redeem_rental" change the same attribute "bonus". (See the pre- and post-conditions in Fig. 4(9) and imagine corresponding rules for the instantiation of "Travel" to "Flight" and to "Rental", respectively.)

Conversely, if p_B depends on p_A, this is equivalent to a conflict between ${p_A}^{-1}$ and p_B. Therefore, a hypothetical application of A^{-1} would make p_B inapplicable, i.e., at least one of the constraints imposed by the left-hand side of p_B could not be satisfied when all the constraints of the left-hand side if A hold. If, in addition, A does not impede B, this means that immediately after the execution of A some preconditions of B that were false before A execution are currently true, i.e., A *enables* B. For example, applying "Book_flight" to a host graph without a "books" edge makes the rule "Pay_flight" potentially applicable. Instead, "Pay_flight" does not enable "Book_leg", since—while it had a potential dependence—it also conflicts with it.

5.1 Analysis of Composition

Impediments and enablements can be used to better understand the coherence of the system specification. In this section, we suggest some constraints that should be satisfied by any consistent system even if some of them cannot be checked completely automatically. Our heuristics assume that the composed activities work in the same context, i.e., the pre- and post-conditions use the same entities, i.e., nodes in the objects graph. In general, a potential conflict might not lead to a concrete conflict; this is especially true in the case of change/use conflicts which often indicate a collaboration: an activity uses attributes changed by another activity. As in Sect. 4, we consider activities connected by flow, fork, join, decision, and merge edges. We adopt the following semantics:

- A `flow` B: the post-condition of A holds before B starts;
- A `fork` B, C: the post-condition of A holds before B and C start;
- A `decision` B, C: the post-condition of A and the decision condition hold before B starts; the post-condition of A and the negation of the decision condition hold before C starts;
- A, C `join` B: the post-conditions of A and C hold before B starts (i.e., A and C act in parallel and they do not interfere);
- A, C `merge` B: merge edges are normally used to resume sequential flow after the branching decision. The post-condition of A or C holds before B starts: i.e., we do not know which one between A and C finishes before B.

A `flow` between A and B is possible only if A does not impede B. Similarly, A `fork` B, C is possible only if A does not impede B and C. A `decision` is analogous to a `fork` when—as it is in most cases—the condition is not on matching constraints; otherwise one should consider the conjunction between the post-conditions of A and the condition decision (or its negation). A, C `join` B is in general possible only if A does not impedes B and C does not impede B. A, C `merge` B is likely to have some problems if A impedes B or C impedes B: however, the branching condition could resolve the conflict in some cases.

Aspect-oriented composition creates new `flow` edges in the system. If more than one piece of advice insists on the same joinpoint which are not in conflict, one should safely consider them as in `fork-join` relation. If, instead, they interfere, the designer should specify deterministically the order of composition. Let J be the joinpoint activity where an aspect activity A is advised, I its direct predecessor and K its direct successor; the heuristics for the weaving operations described in Sect. 2.2 is the following:

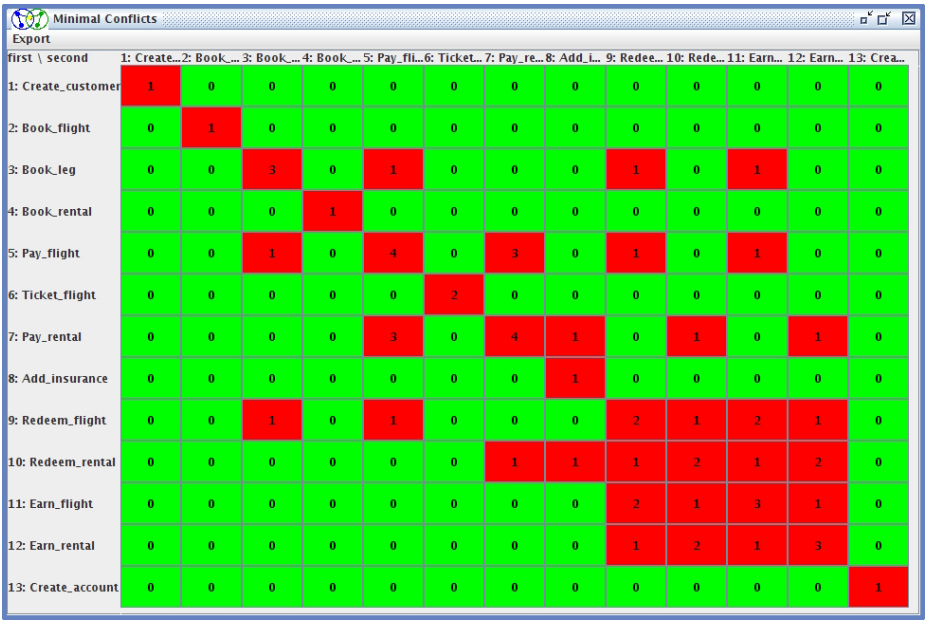
- For `before` J, A must not impede J and must not be impeded by I. If I enabled J, the same enablements should be provided by A.
- For `after` J, A must not be impeded by J and they must not impede K. If J enabled K, the same enablements should be provided by A.
- For `replace` J, A must not be impeded by I and must not impede K. If J enabled K, the same enablements should be provided by A.
- In all cases, the aspect activities may only be impeded by indirect predecessors if other indirect predecessors provide corresponding enablements after disablements. Similarly, aspect activities may impede indirect successors if these are enabled by other successors. Aspect activities do not have to be completely enabled by the use case with which they are composed.

6 Analysis of the Travel Agency Example

In the previous sections, we introduced graph transformation as the theoretical foundation for detecting conflicts and dependencies between activities specified graph with pre- and post-conditions. We computed automatically all potential conflicts and dependencies for the travel agency example using AGG. The results are presented with a conflict (Fig. 13) and a dependency (Fig. 14) matrix. The

first column and first row contain the list of all rules. The entry is a number specifying how many different conflicts/dependencies (critical pairs) were found. An entry in light color (green) contains a zero. An entry in dark color (red or blue) contains a number different from zero; here, it can range from one to four.

- *Conflict matrix:* a positive entry indicates that column entry A *disables* row entry B; B is in conflict with A.
- *Dependency matrix:* a positive entry means that column entry A is a *dependency* for row entry B; B is dependent on A.



first \ second	1: Create...	2: Book_...	3: Book_...	4: Book_...	5: Pay_fli...	6: Ticket...	7: Pay_re...	8: Add_i...	9: Redee...	10: Rede...	11: Earn...	12: Earn...	13: Crea...
1: Create_customer	1	0	0	0	0	0	0	0	0	0	0	0	0
2: Book_flight	0	1	0	0	0	0	0	0	0	0	0	0	0
3: Book_leg	0	0	3	0	1	0	0	0	1	0	1	0	0
4: Book_rental	0	0	0	1	0	0	0	0	0	0	0	0	0
5: Pay_flight	0	0	1	0	4	0	3	0	1	0	1	0	0
6: Ticket_flight	0	0	0	0	0	2	0	0	0	0	0	0	0
7: Pay_rental	0	0	0	0	3	0	4	1	0	1	0	1	0
8: Add_insurance	0	0	0	0	0	0	0	1	0	0	0	0	0
9: Redeem_flight	0	0	1	0	1	0	0	0	2	1	2	1	0
10: Redeem_rental	0	0	0	0	0	0	1	1	1	2	1	2	0
11: Earn_flight	0	0	0	0	0	0	0	0	2	1	3	1	0
12: Earn_rental	0	0	0	0	0	0	0	0	1	2	1	3	0
13: Create_account	0	0	0	0	0	0	0	0	0	0	0	0	1

Fig. 13. Conflict Matrix in AGG

By using these matrices one can reason about impediments and enablements. In the conflict matrix, each activity is in at least one conflict with itself, which is typical for changes effected once. They can be ignored in the following. Many conflicts and dependencies that are caused by changes to attributes will not be considered in the following since they do not cause any effect. The matrices can be used for validating the internal consistency of a single use case. For example, "Add_insurance" is impeded by "Pay_rental" but never occurs thereafter in use case "Buy flight".

Because of flattening the typing relation, we have to look at four concretely typed compositions. We describe results related to flights; rentals are similar. For ease of analysis a graph depicting impediments and enablements can be used (Fig. 16). It contains a directed edge A to B if A impedes (solid red) B or A enables B (dashed blue). Undirected edges represent mutual conflicts. AGG is

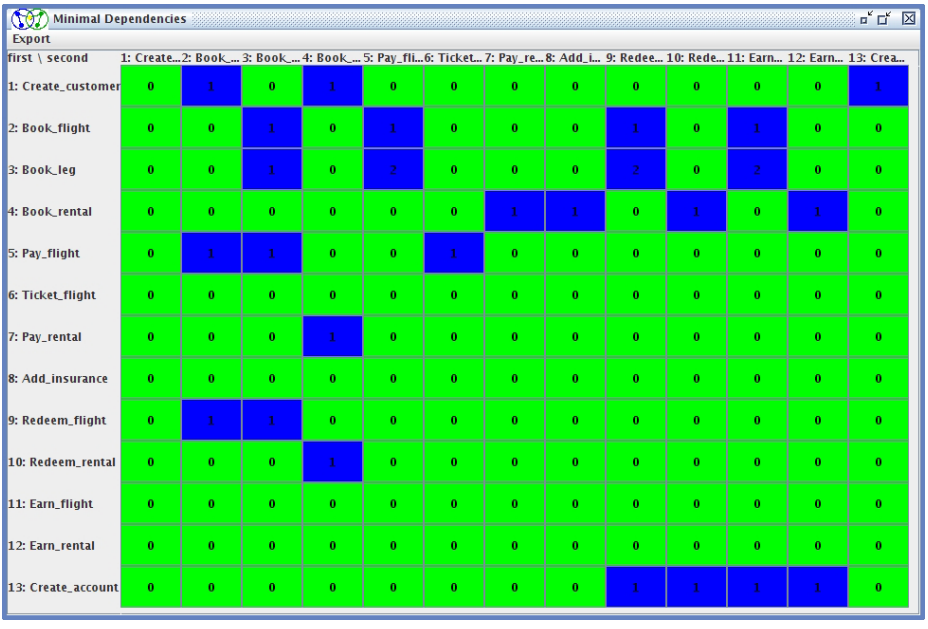


Fig. 14. Dependency Matrix in AGG

able to generate automatically the graph of conflicts and dependencies and the user may hide irrelevant edges or nodes in order to focus on specific concerns.

Before-Composition: Consider the use case "Earn_bonus" (Fig. 3c) `before` any "Pay" activity. This use case contains "Earn_flight" (flattened) and "Create_account". "Earn_flight" is impeded by "Pay_flight" but never occurs after it. After the composition, "Earn_flight" could happen after a "Book_leg" activity that has a potential impediment on it; however, the study of the critical pair between "Earn_flight" and "Book_leg" (see Fig. 15) reveals that the conflict is not critical (it is in fact desired: earning the bonus is influenced by cost of the flight). "Pay_flight" is not impeded by "Earn_flight". "Earn_flight" is impeded by "Redeem_flight", which is, however, not part of the composed use cases. "Pay_flight" was enabled by "Book_flight", and it still is, since "Earn_flight" does not interfere with the "book" edge added by "Book_flight".

"Earn_flight" is enabled by "Book_flight" and occurs after it. "Earn_flight" has also a dependency on "Book_leg" (see Fig. 14). Thus, a bonus is earned for the flight and each leg. This inconsistency is a kind of jumping aspect problem [27]. A negative condition can be added to prevent earn from being applied to a leg. "Create_account" is enabled by "Create_customer" that occurs before it.

Replace-Composition: Consider the use case "Redeem_bonus" (Fig. 3c) that `replaces` "Pay" activities. This use case contains "Redeem_flight" (flattened). "Pay_flight" impedes "Redeem_flight" and vice versa. Using the graph, one can easily find activities depending on "Pay_flight". "Ticket_flight" is dependent on "Pay_flight" but not on "Redeeming_flight". "Redeem_flight" states its

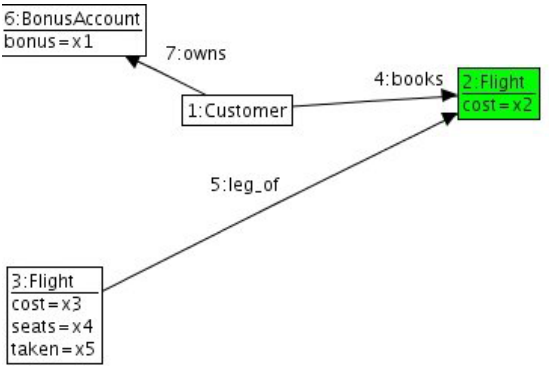


Fig. 15. Change/use conflict between "Earn_flight" and "Book_leg"

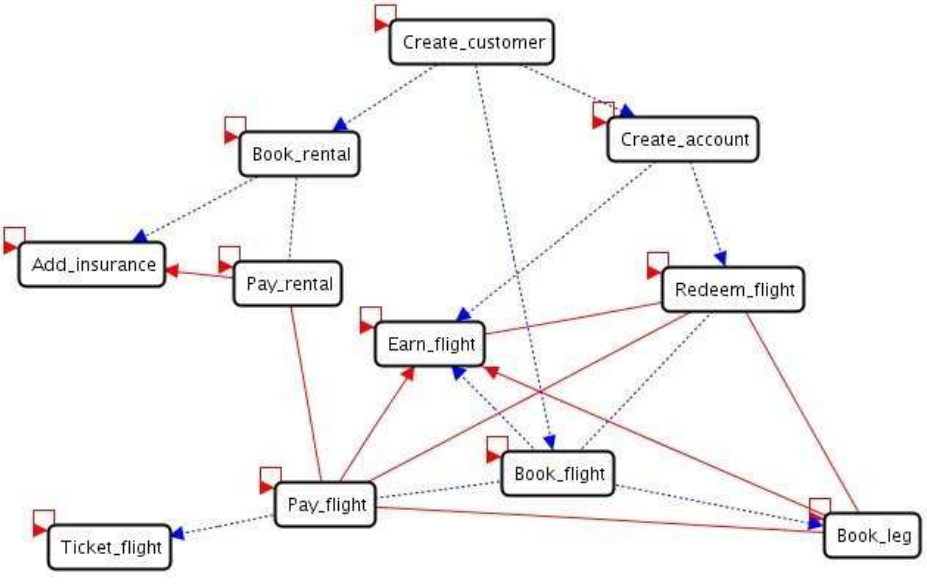


Fig. 16. Impediments (solid red) and enablements (dashed blue)

post-condition in its own domain, i.e., it inserts an edge "redeems". Note that this is a semantic conflict although there is no edge corresponding to it in the graph.

"Redeem_flight" depends on "Book_flight" and "Book_leg". It requires only an edge "books". Thus, the bonus is paid for the flight and each leg, which is undesirable as discussed above. Also, "Redeem_flight" depends on "Create_account", an activity outside both use cases.

The problem that redeem disables ticketing has its origin in the co-existence of the paying domain and the bonus system. A quick fix would be that redeem

also state side effects in terms of the paying domain of flight booking. In its post-condition an edge "pays" could be inserted. A more sophisticated and more maintainable solution would be to keep redeem free of the paying. Instead, an adaption could be added to the ticketing, in order to enable ticketing for redeemed flights. Here, a new aspect could specify the production of a ticket that has no price but to the bonus that was used to redeem it. This aspect would have to be used as a replace aspect.

Applying a heuristic such as the last one can turn out to be a hard problem. The general question is whether the overall activity diagram resulting from the weaving conforms with the overall conflict and dependency graph. In fact, conditional branching and cycles in activity diagrams (and in the graph) make it impossible to determine statically every enabling or disabling action. AGG provides formal support to our approximate solution of this hard problem.

7 Related Work

Conflict analysis based on graph transformation has been applied in several contexts in software engineering. The detection of conflicts in functional requirement specifications was investigated in [23], where we considered requirement specifications developed with the use case-driven approach. The motivation of this work was the early detection of conflicts within the software engineering process. Another application in this area is the detection of conflicts and dependencies in software evolution, more precisely between several software refactorings [28]. Both investigations use the critical pair analysis of AGG for detection. Closely related to this work is the stratification approach for relational graph rewrite systems in [29]. Considering rewriting of labeled graphs by a special kind of rules, different kinds of dependencies are identified and summarized in a dependency graph, similar to our approach. In contrast to the graph rewriting approach in [29] based on labeled graphs, we use the algebraic approach to graph transformation for typed, attributed graphs. For the analysis of aspect interactions, we use the comprehensive theory of algebraic graph transformation summarized in [11]. Especially useful is the Critical Pair Lemma which states that local confluence of all graph transformations can be gained if all critical pairs (presenting conflicts in minimal context) are local confluent.

In our approach, we use graph transformation twice: First for formally defining the semantics of pre- and post-conditions for activities and second on the metamodeling level to precisely define aspect weaving on the basis of activity diagrams. This second way has already been followed by Aßmann [30] in a similar approach. The conflict and dependency analysis is performed on graph rules of the first kind only, since they are the ones which present the formal semantics of aspects. Whittle et al. [31] use graph transformations to weave together behavioral UML models (expressed by using a Role-Based Metamodeling language): behaviors can be composed as alternatives, parallel interleaving and plain sequence.

A clustering of individual requirements within the specification of the behavioral characteristics of a system is often called a *feature* [32]. The notion

of feature, while natural in the "problem domain", is not always present in the "solution domain". In fact, researchers in *feature engineering* propose promoting features as "first-class objects" within the software process in order to bridge the gap between the user needs and design or implementation abstractions. However, in general, features are not independent of each other nor necessarily consistent. Finkelstein et al. [33] proposed a framework for tracking relationships among different *viewpoints* of a system, according to the goals pursued by the different stake-holders involved in the system development. By contrast, our analytical approach aims to detect inconsistencies and interactions as early as possible in order to avoid them.

In [2], Araújo et al. describe non-aspectual requirements as sequence diagrams and aspectual requirements with interaction pattern specifications, which are both woven together in state machines that can be simulated. No support for static conflict detection is provided.

Nakajima and Tamai [34] use Alloy [35] to analyze interactions among role models, taking into account object-oriented refinement and aspect-oriented weaving. A similar approach is taken by Mostefoaoui and Vachon [36] who use an aspect-oriented extension of UML to model the system under analysis. Alloy is able to check properties of relational models by exploiting SAT solving algorithms. The tool enumerates coherent problem instances (under a given scope assumption) and search for counterexamples that disprove the desired properties. Critical pair analysis, instead, focuses only on graph transformation rules: specific problem instances are abstracted away and therefore are more suitable to be used at requirement level and during the very early stages of development.

Several researchers are working to find interactions at the programming level, normally in AspectJ code. Specific program analysis techniques for AspectJ programs were proposed [9, 10] in order to determine whether two aspects interfere. Clifton and Leavens [37] propose classifying aspects in *observers,* which do not change the system behavior, and *assistants,* which participate actively in the global computation. Similarly, Katz [7] proposed using data-flow analysis to identify *spectative, regulative,* and *invasive* aspects. These techniques can be used to automatically extract models of the code, which can be used to verify that expected properties of the system hold [8, 38–40]. Douence et al. [41] introduced a generic framework for aspect-oriented programming supporting pointcuts with explicit states, and they provided an abstract formal semantics of their aspect language: this allows the detection of aspect interference.

8 Conclusion

A key problem of aspect-oriented composition is the use of quantification, which makes it more difficult to reason about than in purely object-oriented models. On the other hand, identifying aspects already on the requirements level and supporting them in the sequel throughout the entire life cycle makes it easier to understand and analyze desirable properties of a system. They can be easier conceived and formulated than in a tangled system.

In this paper, we present an approach for detecting conflicts and dependencies in behavioral specifications of aspects. It is based on use cases refined by activity diagrams. We use pre- and post-conditions for activities to make the modeling more rigorous. Specifying pre- and post-conditions may require an additional effort but pays off if an early formal analysis is required. We think that this is the case when developing frameworks or product lines with aspect-oriented techniques. In this situation, the behavior of modules is specified more rigorously and potential compositions of aspectual and non-aspectual components can be checked in advance with the benefit that for a larger number of products the consistency has been analyzed. The number of activities used to refine use cases highly depends on the process and the context in which the software is developed. However, the analysis of pre- and post-conditions is not restricted to activities, but can also be applied to methods and to a wide range of aspect-oriented modeling techniques if they are enhanced by pre- and post-conditions, which are a universally applicable technique.

In our approach, we use graph transformation as a formal technique to give a formal semantics to the chosen UML variant and to analyze it rigorously. Detected conflicts and dependencies are potential ones that might occur in the system but do not have to. Nevertheless, the formal technique helps to make the problems explicit. It directs the developer to the problematic parts of a model. It helps in understanding aspect-oriented compositions and also in reasoning effectively about the crosscutting. Graph transformation also allows us to reason uniformly about object and aspect models.

On the meta level, we use graph transformation to formalize the composition of aspect use cases with base use cases by specifying a set of transformations that weave aspect use cases into the base use cases according to the composition operators. This formalization provides the basis for the interaction analysis as it provides the overall control flow based on which pre- and post-conditions of activities can be compared.

Support for analysis of the conflict and dependency graph is definitely needed in order to apply the ideas to larger examples. The analysis of conflicts and dependencies can be carried out with the tool AGG, a tool for specifying and analyzing rule-based transformations of typed attributed graphs. Since the analysis functions are provided with a Java API, AGG is ideal for use with existing UML CASE tools. Furthermore, incremental analysis would be desirable and feasible in such a setting. AGG can also support the formalization of the weaving.

The presented approach can be applied in two ways. It can be used to validate an aspect-oriented design by comparing operators with conflicts as demonstrated in this paper. It can also be used to propose aspect-oriented compositions by deriving them from conflicts and dependencies.

9 Outlook

To consolidate our approach and to assess its usefulness in practice, we want to apply the interaction analysis to one of the case studies that was conducted in the TOPPrax project [42]. In this case studies, a security product line has been

developed using the aspect-oriented language ObjectTeams/Java [43–45]. The different features that can be chosen for a concrete product line instance can be checked for their compatibility.

While specifying the pre- and post-conditions with object diagrams over domain classes has been advocated by many object-oriented methods [18, 19, 23], the UML primarily proposes another solution. Conditions over model instances are specified using OCL [12]. It is up to future work to analyze aspect interaction on the basis of OCL. One possible solution could be to transform them to graph rules, as presented in [46]. But we have to be aware of the fact that single rules cannot represent all kinds of constraints.

One feature that can be extremely useful for reasoning about aspect-oriented models is unification of types. Often, a reusable aspect model does use its own types which are not necessarily the same as those used in the domain class diagram. Future work on interaction analysis should incorporate type unification facilities as well. Currently, AGG is integrating object-orientation also into the analysis facilities. So far, it has not yet integrated aspect-oriented facilities.

References

[1] Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley, Reading (2005)
[2] Araújo, J., Whittle, J., Kim, D.K.: Modeling and composing scenario-based requirements with aspects. In: Proceedings of the 12th IEEE Int. Requirements Eng. Conf. IEEE Computer Society Press, Los Alamitos (2004)
[3] Rashid, A., Sawyer, P., Moreira, A., Araújo, J.: Early aspects: A model for aspect-oriented requirements engineering. In: Proc. IEEE Joint International Conference on Requirements Engineering, pp. 199–202. IEEE Computer Society Press, Los Alamitos (2002)
[4] Rashid, A., Moreira, A., Araújo, J.: Modularisation and composition of aspectual requirements. In: Proc. AOSD 2002, Enschede, Netherlands, pp. 11–20. ACM Press, New York (2002)
[5] Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness. In: Proceedings of OOPSLA 2000 workshop on Advanced Separation of Concerns (2000)
[6] Sillito, J., Dutchyn, C., Eisenberg, A., DeVolder, K.: Use case level pointcuts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 246–268. Springer, Heidelberg (2004)
[7] Katz, S.: Diagnosis of harmful aspects using regression verification. In: Leavens, G.T., Lämmel, R., Clifton, C. (eds.) Foundations of Aspect-Oriented Languages (2004)
[8] Rinard, M., Sălcianu, A., Bugrara, S.: A classification system and analysis for aspect-oriented programs. In: Proceedings of SIGSOFT 2004/FSE-12, Newport Beach, CA, USA, pp. 147–158. ACM, New York (2004)
[9] Zhao, J.: Slicing aspect-oriented software. In: Proceedings of the 10th IEEE International Workshop on Programming Comprehension, pp. 251–260 (2002)
[10] Balzarotti, D., Castaldo D'Ursi, A., Cavallaro, L., Monga, M.: Slicing AspectJ woven code. In: Proceedings of the Foundations of Aspect-Oriented Languages workshop (FOAL 2005), Chicago, IL, USA (2005)

[11] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in TCS. Springer, Heidelberg (2005)
[12] Object Management Group: UML Specification Version 1.5, formal/03-03-01 (2003), http://www.omg.org
[13] Technische Universität Berlin: AGG Homepage (2007), http://tfs.cs.tu-berlin.de/agg
[14] Mehner, K., Monga, M., Taentzer, G.: Interaction Analysis in Aspect-Oriented Models. In: International Conference on Requirements Engineering RE 2006 (2006)
[15] Herrmann, S., Hundt, C., Mehner, K.: Mapping Use Case Level Aspects to Object Teams/Java. In: OOPSLA Workshop on Early Aspects (2004)
[16] Mehner, K., Taentzer, G.: Supporting Aspect-Oriented Modeling with Graph Transformations. In: AOSD 2005 Workshop on Early Aspects (2005)
[17] Araújo, J., Coutinho, P.: Identifying aspectual use cases using a viewpoint-oriented requirements method. In: Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design, Boston, MA, USA (2003)
[18] Coleman, D.: Object Oriented Development, The Fusion Method. Prentice-Hall, Englewood Cliffs (1994)
[19] D'Souza, D., Wills, A.: Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, Reading (1998)
[20] The Eclipse Foundation: AspectJ Homepage (2007), http://www.eclipse.org/aspectj/
[21] Katz, S.: A Survey of Verification and Static Analysis for Aspects (AOSD-Europe Network of Excellence (2005), http://www.aosd-europe.net
[22] Plump, D.: Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In: Term Graph Rewriting – Theory and Practice. John Wiley and Sons, Chichester (1993)
[23] Hausmann, J., Heckel, R., Taentzer, G.: Detection of Conflicting Functional Requirements in a Use Case-Driven Approach. In: Proc. of Int. Conference on Software Engineering 2002, Orlando, USA (2002)
[24] Ermel, C., Taentzer, G., Bardohl, R.: Simulating Algebraic High-Level Nets by Parallel Attributed Graph Transformation: Long Version. Technical Report 2004-21, Technische Universität Berlin (2005)
[25] Taentzer, G.: Parallel high-level replacement systems. TCS 186 (1997)
[26] Taentzer, G.: Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems. PhD thesis, TU Berlin. Shaker Verlag (1996)
[27] Brichau, J., De Meuter, W., De Volder, K.: Jumping aspects. In: Workshop Aspects and Dimensions of Concerns, ECOOP 2000, Sophia Antipolis and Cannes, France (2000)
[28] Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts unsing Critical Pair Analysis. In: Heckel, R., Mens, T. (eds.) Proc. Workshop on Software Evolution through Transformations (SETra 2004), Satellite Event of ICGT 2004, Rome, Italy. ENTCS. Elsevier, Amsterdam (2004)
[29] Aßmann, U.: Graph rewrite systems for program optimization. Transactions on Programming Languages and Systems 22(4), 538–637 (2000)
[30] Aßmann, U., Ludwig, A.: Aspect Weaving with Graph Rewriting. In: Czarnecki, K., Eisenecker, U.W. (eds.) GCSE 1999. LNCS, vol. 1799, p. 24. Springer, Heidelberg (2000)

[31] Whittle, J., ao Araújo, J., Moreira, A.: Composing aspect models with graph transformation. In: EA 2006: Proceedings of the 2006 international workshop on Early aspects at ICSE, Shanghai, China, pp. 59–65. ACM Press, New York (2006)
[32] Turner, C.R., Fuggetta, A., Lavazza, L., Wolf, A.L.: A conceptual basis for feature engineering. The Journal of Systems and Software 49(1), 3–15 (1999)
[33] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A framework for integrating multiple perspectives in systems development. International Journal of Software Engineering and Knowledge Engineering 1(2), 31–58 (1992)
[34] Nakajima, S., Tamai, T.: Lightweight formal analysis of aspect-oriented models. In: UML 2004 Workshop on Aspect-Oriented Modeling (2004)
[35] Jackson, D.: Alloy: a lightweight object modelling notation. Software Engineering and Methodology 11(2), 256–290 (2002)
[36] Mostefaoui, F., Vachon, J.: Verification of Aspect-UML models using Alloy. In: Aspect Oriented Modelling Workshop, Vancouver, British Columbia, Canada, pp. 41–48. ACM press, New York (2007)
[37] Clifton, C., Leavens, G.T.: Obliviousness, modular reasoning, and the behavioral subtyping analogy, Technical Report TR03-01a, Iowa State University (2003); presented at SPLAT 2003
[38] Xu, W., Xu, D.: A model-based approach to testing interactions between aspects and classes. In: AOSD 2005 Workshop on Testing Aspect-Oriented Programs, Chicago (2005)
[39] Ubayashi, N., Tamai, T.: Aspect-oriented programming with model checking. In: AOSD 2002 (1st International Conference on Aspect-Oriented Software Development) Conference Proceedings, Enschede, NL, pp. 148–154 (2002)
[40] Denaro, G., Monga, M.: An experience on verification of aspect properties. In: Tamai, T., Aoyama, M., Bennett, K. (eds.) Proceedings of the International Workshop on Principles of Software Evolution IWPSE 2001, Vienna, Austria, pp. 184–188. ACM, New York (2001)
[41] Douence, R., Fradet, P., Südholt, M.: Composition, reuse, and interaction analysis of stateful aspects. In: Proceedings of the 3rd international Conference of Aspect-oriented Software Development, Lancaster, UK. ACM, New York (2004)
[42] Project, T.: TOPPrax home page (2007), http://www.topprax.de
[43] Herrmann, S.: Object Teams homepage (2007), http://www.ObjectTeams.org
[44] Sokenou, D., Mehner, K., Herrmann, S., Sudhof, H.: Patterns for Re-usable Aspects in Object Teams. In: NODe 2006, at Net. ObjectDays 2006, Erfurt (2006)
[45] Hundt, C., Mehner, K., Pfeiffer, C., Sokenou, D.: Improving Alignment of Crosscutting Features with Code in Product Line Engineering. Journal of Object Technology Special Issue of TOOLS Europe 2007, Zurich (2007)
[46] Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: Consistency Checking and Visualization of OCL Constraints. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 294–308. Springer, Heidelberg (2000)

Author Index