Towards expressive, well-founded and correct Aspect-Oriented Programming

Mario Südholt

Rapport scientifique

présenté pour obtenir une

Habitation à diriger des recherches

de l'université de Nantes.

Spécialité Informatique

Towards expressive, well-founded and correct Aspect-Oriented Programming

Mario Südholt

Projet OBASCO

École des Mines de Nantes – INRIA

Laboratoire d'Informatique de Nantes Atlantique (LINA)

Soutenue le 11 juillet 2007 devant la commission d'examen composée de

Charles CONSEL	Prof. à l'ENSEIRB, Bordeaux	Rapporteur
Karl LIEBERHERR	Prof. à la Northeastern University, Boston	Rapporteur
Jan VITEK	Prof. à la Purdue University, West-Lafayette	Rapporteur
Frédéric BENHAMOU	Prof. à l'université de Nantes	Président
Thomas JENSEN	Directeur de recherches, CNRS, Irisa, Rennes	
Pierre COINTE	Prof. à l'école des mines de Nantes	

Acknowledgments

The lion's share of the results reported on in this thesis has been achieved within OBASCO project at École des Mines de Nantes, to a large part by means of different collaborations involving almost all of my colleagues there. These are acknowledged in the text, as some cooperations with other researchers, as the corresponding work is presented. Here, I would just like to express my gratitude to all of them for having contributed the competences, advice and effort necessary for significant work in a research field that covers much of software engineering as well as related research and application domains.

I am much indebted to Pierre Cointe, head of OBASCO project, who has frequently provided me with strategic advice how to orient my research and consistently insisted on me taking on this habilitation work.

Special thanks go to Daniel Le Metayer who welcomed me on my arrival in France in Lande group at INRIA/IRISA institute in Rennes and Pascal Fradet who raised and fostered my interest in the rigorous treatment of aspects there. I am also grateful to Claude Labit, director of INRIA-Rennes, for having supported my successful bid for a two-year secondment at INRIA that has provided the necessary leeway to initiate the more recent research directions I am glad to present here.

Some of my work has been the result of cooperations or influenced by intense discussions with different foreign researchers. Wim Vanderperren, Theo D'Hondt and Viviane Jonckers from SSEL and PROG groups at Vrije Universiteit Brussel have contributed significantly to some of the results presented here and to my views on the associated research issues. Jan Vitek from Purdue University has always been a good friend whose support and different angle of view on software engineering issues and research in general has proved very helpful over the years.

Finally, my most affectionate thanks go to Maël, Thomas and Manue who, in particular, granted me much more of their patience during the last three months than I probably should have asked for.

Preface

This habilitation thesis presents the main results of my research work conducted as part of OBASCO ("Objects, Aspects, Components") project, a joint project with INRIA that is also part of the "Laboratoire d'Informatique de Nantes Atlantique" (LINA), in the computer science department of École des Mines de Nantes. This work has focused on modularization and software evolution problems investigated as part of the emerging research domain of Aspect-Oriented Programming, in particular, the foundation and design of aspect languages, the relationship between component-based programming and aspects, as well as aspect-oriented support for distributed programming.

Similar problems had already been the subject of my previous work. As part of my PhD thesis at TU Berlin I investigated the parallelization of functional programs by means of a skeleton-based approach. This work can, in today's terminology, easily be framed as the definition of several domain-specific aspect languages for parallel execution at different levels of abstraction, ranging from a high-level language to languages directly representing architectures of parallel machines. I have also provided an implementation in terms of a calculus allowing the transformation between abstraction levels, a method very similar to aspect weaving.

In my following year of post-doc studies at IRISA/INRIA-Rennes, I employed shape types — a notion of graph-based types initially developed to provide better structured support for data types in the C language — to devise new abstraction and reasoning mechanisms for software architectures. Such graph-based types allow, in particular, to formally define software architectures underlying typical component-based systems and, based on a notion of refinement, model dynamic evolution of these systems.

On my arrival at OBASCO group in 1997, I started working on reflective object-oriented systems, but became rapidly interested by the subject domain I report on in this habilitation: the lack of means for the modularization of so-called crosscutting functionalities as a fundamental problem for the engineering of large-scale applications. Originating at Xerox PARC within the group headed by Gregor Kiczales (but more generally influenced by the reflection programming community), this problem has sparked in 1996 the development of a new style of programming featured by the emerging research domain of Aspect-Oriented Programming [Kic96, KLo97, ACEF04].

My studies at OBASCO group, which are presented in this document, focused on three fundamental problems with then and current aspect languages. First, most aspect languages use what is called the "atomic pointcut model" in this thesis, i.e., only allow pointcuts to denote sets of unrelated execution events. This model, however, frequently leads to verbose and difficult to understand aspect definitions. The main thread running through my work has been the quest for aspect languages which allow the direct formulation of relations between execution events and thus support the concise formulation of complex aspects. Second, the first full-fledged aspect languages, AspectJ being the best known among them, for a long time lacked formal semantics and did not (and for most still do not) support reasoning about aspect-oriented programs.

I have been interested from the start in providing a foundation of aspect languages by means of

precise semantic frameworks, in particular, striving for a notion of “property-based” AOP. Furthermore, while the main application domain of AOP is, in principle, that of distributed programming, the large majority of AOP approaches study aspect languages in the context of sequential programming. Most recently, in the context of the European Network of Excellence “AOSD-Europe”, I have therefore aimed at explicit support for concurrent and distributed aspect-oriented programming. Finally, while most of my work on aspects focuses on aspect language design issues, almost all of this research work has been employed in different application domains, ranging from standard sequential crosscutting concerns, via concerns at the level of systems programming and operating systems, to concerns of large-scale distributed applications, such as cache replication for distributed component-based systems. This document also presents some of the main characteristics of these application-related results.

Besides aspect-oriented programming, component-based systems constitute a second focus of my work. More precisely, I have investigated notions of components with explicit protocols in their interfaces. By means of protocols of different levels of expressiveness a wide range of component properties can be captured: interactions among components, in particular, can be defined precisely and analyzed. Moreover, as shown in this thesis these features also apply to component-based systems that are subject to modification by aspects.

Contents

Chapter 1

Introduction

A key insight of recent software engineering research is that crosscutting functionalities — i.e., functionalities that cannot be modularized using traditional structuring means, such as objects and components — constitute a major problem for the development of virtually all large software applications. This problem has been identified, for instance, in such different domains as component-based software engineering, feature-based telephone switching networks, operating systems and software product lines. Starting with Dijkstra's [Dij74] and Parnas's [Par72] seminal work that respectively introduced the concepts of "separation of concerns" and "modularization", much research has focused on methods to modularize such functionalities, in particular, based on techniques using metaprogramming and computational reflection. Relatively few of these approaches, however, have provided dedicated language support featuring declarative means for the definition of crosscutting functionalities. Recently, Aspect-Oriented Software Development (AOSD) has emerged as the research field striving for new declarative modularization mechanisms to tackle crosscutting functionalities on all abstraction levels (from software architectures via programs to executing code) as well as during all software development phases (from requirements engineering, via analysis and design to implementation and software maintenance, including adaptation).

AOSD techniques have stirred considerable interest, both in the research community and the software industry. It has, for instance, been included in 2001 in the MIT's list of technologies that "will change the world" [Kic01]. AOSD has proved highly interesting from an academic point of view especially because of the inherent complexity of the definition and implementation of languages for the modularization of crosscutting functionalities, be they used for application design, programming or implementation. Furthermore, AOSD is at the crossroads of a large number of related research domains: notable research work has been done, among others, on its relation to programming language design and implementation, component-based programming, middleware and operating systems. Finally, AOSD techniques are of eminent practical relevance due to the importance of crosscutting functionalities, such as distribution and transactional behavior in, e.g., component-based 3-tier applications.

While research in AOSD has brought into focus crosscutting as a central problem for large-scale software engineering and proposed first solutions to some of the corresponding modularization problems, the available body of work on AOSD and related fields has revealed several problems that have not been addressed satisfactorily by mainstream approaches to AOSD. These problems comprise the following three that are of fundamental nature to the field of AOSD as well as to the larger domain of software engineering:

1. The *level of abstraction* of mainstream aspect languages is too low to modularize many intricate

crosscutting concerns effectively.

2. The relation between *AO abstractions and traditional modularizing mechanisms* has not yet been sufficiently clarified and their synergetic use for the construction of large-scale software systems is an open issue.

3. Modularization problems of *distributed and concurrent programs* have only been tackled rudimentarily.

Before dwelling on the research directions underlying the work presented in this thesis, let us have a closer look at these challenges.

Level of abstraction. Most fundamentally, the current level of abstraction of mainstream languages for AOP impedes the effective modularization of many non-trivial crosscutting concerns.

AO languages are typically defined in terms of pointcuts, which determine the execution points of a base application where aspects modify a software system, and advice, which define how the base application is to be modified. Currently, the predominant model for AOP is based on what we call in this thesis the *atomic pointcut model*. This model is characterized by pointcuts that denote individual execution points or sets thereof. Relationships between execution points that have to be defined over the execution history of a software system cannot be explicitly represented in terms of such pointcuts and have to be expressed using some external notion of auxiliary state.

However, crosscutting functionalities are frequently most naturally expressed in terms of such relationships between different execution points. Consider, for instance, data replication as part of some caching strategy in a distributed application. The point of time when data has to be replicated as well as the data itself typically depend on different previous execution points: the point when data enters the cache obviously, but also previous points pertaining to the control of concurrent executions relevant to this replication action, such as transaction management.

The resulting lack of high-level abstractions has three major baneful consequences:

- The definition of concerns that exhibit non-trivial dependencies among execution events (like the data replication concern mentioned above) cannot be expressed.

- Correctness properties of aspects for such concerns are not reasonably amenable to formalization and automatic reasoning.

- Means for the composition of aspects are lacking and aspect composition is therefore typically left implicit. This is particularly pernicious to the use of aspects as a general program structuring method of large systems that typically rely on many aspects.

Interaction with existing modularization mechanisms. AOSD mechanisms are not intended to supersede existing program structuring mechanisms but to complement them. Since crosscutting concerns, which are scattered over many different parts of an application, pervade traditional module structures, such as software components and Modula-2 like modules, the question arises how to reconcile aspects and traditional approaches to modularization as they seem contradictory at first sight. The integration of these two concepts entails a large set of intricate problems concerning all of the software lifecyle and software development levels.

Lack of support for non-sequential applications. Non-sequential, that is, distributed and concurrent applications, constitute one of the most important classes of applications that can be targeted by AOSD techniques because of their many crosscutting concerns and typically large size. However, there is almost no explicit support for aspects for non-sequential programming. Instead, almost all current approaches to such applications, for example distributed systems built using software components, rely on sequential aspect language that are used to manipulate non-sequential infrastructures. Such approaches only support insufficient modularization of distribution-related concerns because, for example, functionalities that require modifications on different machines have to be implemented using different programs on the different involved machines.

1.1 Research directions

Most of my work over the last seven years has focused on solutions for these three fundamental problems of AOSD and has been aimed at resolving the resulting software engineering challenges. The leitmotiv of my research has been the development of expressive aspect languages and associated formal frameworks to support the concise and correct modularization of intricate crosscutting functionalities in large-scale sequential and distributed software systems.

In general terms, I have initiated and participated in the definition of a large range of aspect systems and languages that provide a *higher level of abstraction*, especially at the pointcut level, as mainstream aspect languages. This work has given rise to several of the first results on the *foundations of AOP* in the form of formal semantics for expressive (non-atomic) aspect systems and automatic reasoning methods about the properties of AO programs. We have focused on two of the most fundamental properties: aspect interaction and aspect applicability properties.

Furthermore, we have explored means for the *integration of aspects and traditional modularization techniques* in the context of component-based programming and systems programming. We have, in particular, proposed more expressive notions of component interfaces that, in turn, have allowed to tackle the integration of software components and aspects.

Finally, I have initiated the development of the currently most comprehensive *aspect system for explicitly distributed programming* as well as participated in the first approach allowing the *coordination of concurrent aspects*.

1.2 Presentation of this thesis

This habilitation thesis presents the main results of this body of work in terms of instantiations of a general model, *Event-based AOP* (EAOP), the first non-atomic model for the definition of expressive aspect languages we have developed in 2000.

This thesis aims at two different goals. First, a *uniform presentation of the major relevant research results* on EAOP-based expressive aspects. We motivate that these instantiations enable aspects to be defined more concisely and provide better support for formal reasoning over AO programs than standard atomic approaches and other proposed non-atomic approaches. Concretely, four groups of results are presented in order to substantiate these claims:

1. The *EAOP model*, which features pointcuts defined over the execution history of an underlying base program. We present a taxonomy of the major language design issues pertaining to non-atomic aspect languages, such as pointcut expressiveness (e.g., finite-state based, turing-complete) and aspect composition mechanisms (e.g., precedence specifications vs. turing-complete composition programs).

2. Support for the formal definition of aspect-oriented programming based on different semantic paradigms (among others, operational semantics and denotation semantics). Furthermore, we have investigated the *static analysis of interactions* among aspects as well as applicability conditions for aspects. The corresponding foundational work on AOP has also permitted to investigate different weaver definitions that generalize on those used in other approaches.

3. Several instantiations of the EAOP model for aspects concerning sequential program executions, in particular, for *component-based and system-level programming*. The former has resulted in formally-defined notions of aspects for the modification of component protocols, while the latter has shown, in particular that expressive aspects can be implemented in a performance-critical domain with negligible to reasonable overhead.

4. Two instantiations of the EAOP model to *distributed and concurrent programming* that significantly increase the abstraction level of aspect definitions by means of domain-specific abstractions.

Moreover, we discuss a number of *perspectives* ranging from technical advances which can be attained directly from the presented results to solutions to fundamental problems of AOP (and the field of software engineering as a whole). We consider, in particular, how the presented approach provides a basis to address three fundamental problems of concern separation:

- The quest for a well-defined notion of "*aspectual modules*" that reconciles aspects, i.e., modules for crosscutting functionalities, with standard notions of modules (or at least modular reasoning).

- A comprehensive framework of *aspects for distributed component-based programming* that fosters a pattern-based approach to the development of distributed applications. This way recent results on the the synergy between AOP and pattern-based program design and implementation for sequential programs can be leveraged to distributed programming.

- Generalize *model-driven engineering* through the use of expressive aspect languages of different level of abstractions.

While this document presents a large part of my work, several results are not be detailed here, mainly because they would distract the reader's attention from the main thread of argument focused on here: the motivation, definition and application of non-atomic aspect languages for the modularization of crosscutting concerns. Most notably, I do not discuss work I have done on computational reflection and advanced concepts on object-oriented languages [3-DS01, 6-DS00b, 7-DS00a, 4-CND$^+$04]. While AOP can be seen as being historically based on reflection and while there are a number of important conceptual, methodological and technical commonalities between both domains, the relationship between the two domains is not essential for the work presented here. Specific links between reflection and AOP are, however, mentioned in the text and the interested reader is referred to [BL04, Lop04] for further information. A second subject of my work that is not discussed in this document is work on aspects for operating systems [5-ÅLS$^+$03, 7-ÅLSM03, 5-MLMS04]. This work features a non-atomic pointcut language defined using the temporal logics CTL that is statically woven using a transformational system. I do not either expose the very first work on a formal definition of AOP as general transformational systems [7-FS98, 7-FS99] that I have pursued with Pascal Fradet in 1998/99. Finally, I won't elaborate on early work on a formal approach to the definition and analysis of software architectures I have participated during my stay at Lande group at INRIA-Rennes [5-HPS97].

Structure of this document

The remainder of this document is structured in five chapters. Chapter 2 makes a case for non-atomic pointcut languages and introduces the EAOP model on which most of the work described here is based. This chapter introduces, in particular, the language design and implementation issues for aspects we are concerned with by presenting a taxonomy of language features for non-atomic aspect languages. In Chapter 3, our work on the foundations of AO languages is presented, covering different formal semantics for AO programs we have devised, properties over AO programs we have considered and corresponding support for property analysis and verification. Chapter 4 introduces a second set of instantiations of the EAOP model geared towards software components and system-level programming. Chapter 5 discusses the extension of the EAOP model to the distributed and concurrent cases: the recent work on explicitly-distributed AO programming as well as a first approach to concurrent aspects that allows to directly coordinate the concurrent execution of aspects and (concurrent) base programs. A conclusion as well as more fundamental (and prospective) perspectives our approach paves the way for are presented in Chapter 6.

Chapter 2

A case for expressive aspect languages

The lack of modularization techniques for crosscutting functionalities, i.e., functionalities that are scattered all over an application and tangled with the code of other functionalities, constitute a fundamental engineering problem for large-scale software systems. AOSD investigates such crosscutting concerns and promises a solution to this problem, essentially through new modularization mechanisms as part of aspect languages.[1] Aspect languages are typically structured in terms of two sublanguages: a pointcut language that defines when or where a crosscutting functionality should modify an underlying base application, and an advice sublanguage that defines how to modify the base. A major goal of AOSD research is the definition of simple but also expressive aspect languages allowing the concise (and hence easy to understand) definition of aspects. However, the until now predominant aspect model, which is embodied by the AspectJ [KHH+01, Asp], focuses on pointcuts that denote individual ("atomic") execution events or sets thereof but does not allow to declaratively relate execution events in terms of the history of execution. This seriously impedes the concise definition of complex crosscutting concerns and the development of correct AO programs (both by means of informal judgement or formal verification techniques).

This chapter presents three contributions which are central to our approach to overcome this fundamental expression and correctness problem of AOSD. First, one of the fundamental contributions of AOSD has been the identification of crosscutting as a fundamental problem of all large-scale applications. However, until now many applications of aspects are of rather simple structure (e.g., the infamous "logging aspect"). We present an analysis of a richly structured, real-world crosscutting problem: distribution and transactional behavior in a large-scale replicated cache infrastructure, JBoss Cache [jbob]. We show that these two concerns are crosscutting in the presence of (traditional) means for modularization and that the concerns are governed by non-trivial relationships. Second, we analyze to what extent mainstream (i.e., atomic) models for AOP are suitable for the modularization of complex crosscutting concerns. We show that these models are subject to serious deficiencies for such an endeavour. As a solution approach we present a taxonomy of advanced features for aspect languages that address these deficiencies. Third, we propose the model of Event-Based AOP as an alternative model for the modularization of complex crosscutting concerns. This is achieved through three distinctive features of the EAOP model: (i) support for expressive aspect languages that allow, in particular, relationships between execution points to be represented directly on the pointcut level, (ii) an explicit notion of aspect composition, and (iii) the possibility to apply aspects to aspects.

[1] The importance of these two basic contributions of AOSD — identification/analysis of crosscutting concerns and definition of suitable aspect languages — has already been at the heart of the initial AOSD papers published by Gregor Kiczales's group at Xerox research, see [Kic96, KLo97].

The results presented in this chapter, in particular, the analysis of atomic pointcut languages and the EAOP model for expressive aspect languages have been developed as part of the FP5 European IST project on software composition "EasyComp" [EAS] in cooperation with several other researchers and students. Most notably, my colleague Rémi Douence and me have started the work on non-atomic relationships for aspect languages in 2000 and developed the EAOP model. Luis Daniel Benavides Navarro and me have performed the original study of crosscutting concerns in replicated caches [5-BNSV+06a], an extended version of which is presented in this text.

The remainder of this chapter is structured as follows: Section 2.1 presents the analysis of crosscutting concerns in a large-scale infrastructure for replicated caching. Section 2.3 discusses the mainstream (atomic) models for aspect languages in more detail, followed in Section 2.4 by the presentation of the model of Event-based AOP that forms the basis of our work. Section 2.5 gives a conclusion and presents some perspectives.

2.1 Motivation: modularization of complex crosscutting concerns

We first turn to the problem of crosscutting concerns that lies at the heart of AOSD. One of the main motivation of our work has been that, despite its fundamental character, crosscutting has been approached in a shallow manner only. Most work on AOP considers only simple crosscutting concerns as exemplified by numerous articles considering aspects like logging or execution tracing. Relatively few researchers have considered more intricate crosscutting relationships. A notable exception are results relating to AspectJ-style control flow relationships between execution events relevant for crosscutting. Following this idea, Coady et al. [CKFS01, CK03] investigate prefetching in file systems expressed in terms of chains of nested calls between different software levels of an operating system. Finally, almost all work is focused on single aspects, even if aspects are frequently subject to interactions among one another.

In this section, we consider crosscutting in such a general setting. Concretely, we extend one of our previous studies on the modularization of distributed systems with the AWED system [5-BNSV+06a] on distribution and transactions in a industrial-strength replicated cache, JBoss Cache [jbob]. Here we present (partially new) results showing that general relationships between different execution events are crucial for the concise definition of aspects modularizing complex crosscutting concerns and that refactoring of the underlying object-oriented application is not able to resolve this kind of modularization problem. More concretely, our analysis of JBoss Cache contributes to these two fundamental characteristics of crosscutting as follows:

- *Type of relationships governing crosscutting.* Crosscutting generally does not consist of unrelated scattered and tangled actions but of actions that have to be executed by respecting (frequently implicit) protocols. Hence, they cannot be reasonably described in terms of individual execution events but only by making explicit the relationships between different events. Replication and transactional behavior in replicated caches, as shown here for JBoss Cache, is nicely illustrating this: both concerns are highly scattered, tangled with one another, and protocols, e.g., involving the initialization and later use of caches and transactions, are crucial to their correct use.

- *Resistance to refactoring.* While it is well-known that all functionalities of large software systems cannot be modularized at the same time using traditional program structures (see the notion of the "tyranny of the dominant decomposition" introduced by Tarr et al. [TOHS99]), it is an

open question whether the modular structure of existing crosscutting functionalities can be reasonable improved by refactoring using traditional programming means. The JBoss Cache code base has undergone substantial restructuring from March 2005 (version 1.2.1, which is the version analyzed in our previous publication [5-BNSV+06a]) to August 2006 (the current version 1.4.0, which is considered here). This restructuring is partly targeted at a better modularization of the transaction concern. However, we show below that crosscutting of replication and transactional behavior still is an issue in the latest version, thus providing evidence that crosscutting is resistant to restructuring of the underlying code base in large applications.

Analysis of replication and transaction concerns. JBoss Cache is a replicated cache which allows to install caches onto a set of machines, a "cluster", in a distributed Java-based system among which data is to be replicated. In its current version it supports three different caching strategies: (i) no replication, (ii) replication among all machines in the cluster, and (iii) replication among a subset of the cluster ("buddy replication"; this last strategy was not part of the JBoss Cache version studied in our previous work). JBoss Cache is a large application: it is implemented as an object-oriented framework of a little over 50,000 lines of code (LOC) and consists of 16 main packages including about 250 classes.

Figure 2.1[2] shows elements of a crosscutting analysis for two of the main parts of the framework: the class `TreeCache` (see Fig. 2.1a) that implements the tree data structure stored in the replicated cache nodes and the `interceptors` package (see Fig. 2.1b) that is used as an OO abstraction to separate functionalities in the code. Both parts are quite large: the `TreeCache` class, version 1.4, counts over 6,300 LOC, the `interceptors` package contains 33 classes with a total of more than 7,000 LOC.

We have considered five different functionalities that crosscut the JBoss Cache code base (These are colored in the figure according to the color coding defined in Fig. 2.1c.). Three of these crosscutting concerns are part of the replication and transactional behavior. The remaining two relate to the use of interceptors within JBoss Cache. We consider the latter to evaluate the effectiveness of standard OO modularization mechanisms in the presence of crosscutting concerns. Concretely, Fig. 2.1 shows the code distribution of the following functionalities[3]:

1. Distribution, more precisely use of the JGroups library for multicast communication between caches [JGr].

2. Buddy replication, i.e., replication to subsets of clusters (even if buddy replication is part of the replication code we present it separatedly because this functionality is new to the latest JBoss Cache version).

3. Transactional behavior, i.e., calls to the J2EE transaction service.

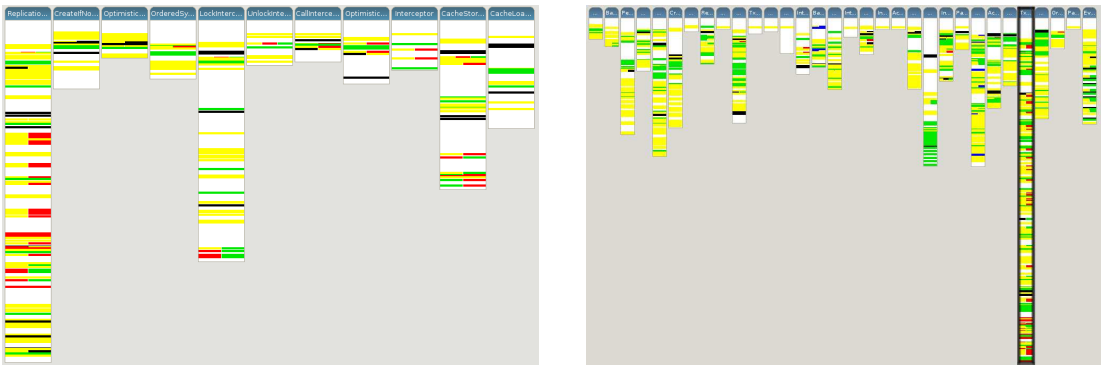
4. Calls between different classes within the `interceptors` package.

5. Calls from the `interceptors` package to the `TreeCache` class.

[2]The illustrations have been generated using the AspectJ plugin 1.4 for the Eclipse platform, version 3.2.

[3]Note that Fig. 2.1 represents a lower bound of the corresponding code scattering: all colored code parts are part of the corresponding functionalities but there are instructions part of the functionalities that have not been colored. In fact, the corresponding approximation has been easier to generate, is sufficient to support our claims and is tight enough to provide an accurate picture of the concerns.

(a) class `TreeCache`, versions 1.2 (left) and 1.4 (right)



(b) package `interceptors`, versions 1.2 (left) and 1.4 (right)

- Distribution
- Buddy replication (1.4 only)
- Transactions
- Interceptor calls (`interceptor` package only)
- Calls into `TreeCache` (`interceptor` package only)

(c) Aspects and color coding

Figure 2.1: Crosscutting in JBoss Cache versions 1.2 (left) and 1.4 (right)

```
protected void _createService() throws Exception
{
   if (this.tm_lookup == null && this.tm_lookup_class != null)
   {
      Class clazz =
          Thread.currentThread().getContextClassLoader().loadClass(this.tm_lookup_class);
      this.tm_lookup = (TransactionManagerLookup) clazz.newInstance();
   }


   ...
   // build interceptor chain
   interceptor_chain = new InterceptorChainFactory().buildInterceptorChain(this);
   ...
   switch (cache_mode)
   {
      ...
      case REPL_SYNC:
         ...
         JmxConfigurator.registerChannel(
            channel, server, "JGroups:channel=" + channel.getChannelName() , true);
         ...
}
```

Figure 2.2: Excerpt of method `TreeCache._createService`

```
public Object put(Fqn fqn, Object key, Object value) throws CacheException
{
   GlobalTransaction tx = getCurrentTransaction();
   MethodCall m = MethodCallFactory.create(MethodDeclarations.putKeyValMethodLocal,
                                           new Object[]{tx, fqn, key, value, Boolean.TRUE});
   return invokeMethod(m);
}
```

Figure 2.3: Method `TreeCache.put`

The left-hand sides of Figs. 2.1a and 2.1b reprint the illustrations of our analysis of JBoss Cache version 1.2 [5-BNSV+06a], while the right-hand sides show the corresponding illustrations for version 1.4. These clearly suggest extensive crosscutting with respect to the five considered functionalities for both versions.

This conjecture is confirmed by a detailed analysis of the code. Figures 2.2 and 2.3 show two key methods that are part of the replication code in class `TreeCache`: the former shows part of the creation protocol of the replication service, the second one of the central methods governing replication when a data element is put in the cache. Both methods also include transaction-related code. This is the case although JBoss Code has been restructured between the old and new version, in particular, to achieve better modularization of these crosscutting concerns: the latest version includes a new interceptor specifically introduced for transaction-related behavior. This interceptor is highlighted by the boxed column in the illustration concerning the `interceptors` package in the right-hand side of Fig. 2.1b. Note that crosscutting of distribution-related code and transaction-related code is present as well in this new interceptor as in the remaining classes (albeit, for the transaction concern, to a lesser degree than for JBoss Cache version 1.2).

```
all aspect CollaborativeCachePolicy {
    ...
    pointcut getCache(Cache c, String key):
        call(* Cache.get(String)) && host(cacheGroup)
            && target(c) && args(key);
    ...
    pointcut replPolicy(Cache c):
        replP: seq(s0: initCache(c) -> s1
                   s1: getCache(c, k1) -> s2,
                   s2: putCache(c, k2, val) && eq(k1, k2) -> s1);
    ...
    around(Cache c, String key): step(replP, s1) && args(c, key) {
        Object obj = c.get(key);
        if (obj == null) {
            obj = proceed();
            if(obj != null) { c.put(key, obj); cacheMisses++; }
        }
        updateSummaries();
        return obj;
    }
    ...
}
```

Figure 2.4: Aspect-based cooperative cache (excerpt from [5-BNSV+06a])

Since the replication code is tangled with transaction code at the different steps of the replication protocol — such as cache initialization, getting and storing information — the modularization of the two concerns has to take into account their respective protocols and cannot be reasonably done by manipulation of single steps. Figure 2.4 presents, for example, an excerpt of an extension to the JBoss Cache replication strategy by a collaborative cache strategy that is best defined in terms of the (repeated) sequence of the three events mentioned above (see the pointcut on Lines 7– 10).

There is a large number of other crosscutting concerns which require trace-based relationships. Two other noteworthy domains where such relationships have been investigated — albeit only in sequential contexts and without considering the pertinence of aspects over time — are system-level communication protocols and business rules. Communication protocols, such as TCP, frequently crosscut the code of networking applications, such as web caches used on the Internet [5-DFL+05]. If such protocols are to be modified, a coordinated modification to the behavior of some or all of constituting steps must be implemented. Business rules crosscut application-level enterprise information systems [CD06, 6-DS03]. The corresponding crosscutting concerns typically require to set some initial state, to record information at different execution events, and to apply a modification at a later event. In the case of billing functionality, for example, these execution points comprise product selections by clients, points at which discounts become applicable and the point when the client checks out its purchases [6-DS03].

To conclude this investigation, note that these examples provide concrete evidence for the two general issues of crosscutting relevant for our work. First, complex crosscutting concerns in large-scale applications need protocols to be captured in order to enable their concise modularization. Second, as we have shown for replicated caching, appropriate means for modularization are all the more important because extant crosscutting cannot be fully remedied using traditional mechanisms even through extensive refactoring of the crosscutting code.

2.2 A taxonomy of advanced aspect language features

We now turn to the second fundamental contribution of AOSD: new language-level constructs for the modularization of crosscutting concerns. These constructs are typically structured in terms of aspect languages that consist of a number of sublanguages. These sublanguages include pointcut and advice languages, the best-known abstraction mechanisms of aspect languages, but also sublanguages governing other aspect relationships, such as aspect composition and instantiation of aspects. Most aspect languages only provide support for few such relationships and existing support frequently is limited. The EAOP model, however, has been developed as a comprehensive framework to explore diverse relationships among aspects and between aspects and base applications.

In this section, we first present a taxonomy of advanced language features for aspect languages as a basis for comparison of aspect languages. We illustrate the taxonomy by using it to classify selected existing aspect languages and systems. This taxonomy also allows us to summarize the main characteristics of the AspectJ model of AOP, which is the de-facto standard model of AOP, and to motivate why our EAOP model addresses some drawbacks of the atomic aspect model.

A comparison of aspect languages and models should be based on an existing, comprehensive and largely accepted taxonomy, metamodel or other classification support. However, in the case of aspects there is no such support. Currently all published work that touches on this issue address it only in a very partial manner. Masuhara and Kiczales [MK03b], e.g., provide an abstract framework for the definition of pointcuts and advice in terms of general domains and functions relating them; their approach, however, provides only very coarse-grained means to compare AOP models. Hanenberg et al. [SHU06] provide means for the abstract definition and comparison of different pointcut languages but do not consider other features of aspects.

In contrast, we have chosen to built on an on-going effort to construct a comprehensive metamodel for AOP that is pursued as part of AOSD-Europe, the European Network of Excellence in AOSD [AOS]. An initial version of this model has been released in February 2006 [BMN+06]. More precisely, we augment and refine a taxonomy of aspect language features that is part of that metamodel in order to provide better support for a fine-grained classification of advanced aspect models. We focus on the precise characterization of languages for the definition of pointcuts, advice and aspect compositions, in particular concerning the level of expressiveness they provide. To this end, we provide finer categorizations for the pointcut and advice sublanguage, and introduce four advanced features, in particular aspect composition mechanisms, in the taxonomy.

Our taxonomy is shown in Table 2.1: it consists of six main features each comprising of up to three levels of subcategories (here categories are set in sans serif font (e.g., Pointcut)). In the following we illustrate and discuss the main categories of the taxonomy in some more detail.

Pointcuts

There exists a large variety of pointcut languages that differ from one another mainly with respect to three different characteristics:

- The level of expressiveness of the pointcut language (represented by category Expressiveness in Table 2.1), in particular whether it provides mechanisms to explicitly denote relationships between join points.
- The underlying programming paradigm: object-oriented languages, for instance, partially require other abstractions than logic languages (category Paradigm).

Pointcut	:	– Expressiveness: – Atomic – History: Finite-state, Vpa, Context-free, Turing-complete – Paradigm: Object-oriented, Functional, Logic – Generality: General-purpose, Domain-specific
Advice	:	Same as for feature Pointcut
Composition	:	– Type: – Implicit: Non-deterministic, Deterministic, Undefined – Explicit – Mechanism: – Precedence: Partial, Total – Operator – Program – Object: Aspect, Advice – Scope: All, Stateful – Expressiveness: Finite-state, Turing-complete
State	:	General, Restricted
Instantiation	:	Machine, Class, Object, Cflow, Binding
Activation	:	Init, Arbitrary

Table 2.1: Summary of taxonomy of aspect language features

- Pointcut languages may be designed as general-purpose languages, i.e., supporting the expression of arbitrary aspects, or domain-specific ones (category Generality).

In this text we mostly abstract from the first issue by considering pointcuts expressed over calls, which may equally well denote object-oriented method calls, functional or imperative function calls, and component-based service calls. However, we touch on this issue in the context of the design of pointcut languages for the C language. We also consider several general-purpose as well as domain-specific pointcut languages, in particular for distributed programming.

We focus on an investigation of pointcut languages of different expressiveness (Expressiveness). With respect to this criterion, pointcut languages, or better constructs of pointcut languages, can be classified as Atomic or History-based. An atomic pointcut construct can be defined solely in terms of an individual execution event, while history-based ones must be defined by relating different execution events occurring over time. Note that languages that essentially rely on atomic constructs must have recourse to means external to the pointcut language in order to express such temporal relationships. This is further discussed in Sec. 2.3.

Many different approaches have been put forward to define history-based execution events, to cite just some examples, logic languages [DD99, BMD02], automata-based approaches [5-DFS02b, SVJ03], and grammar-based approaches [WV04]. Adaptive programming [LL04] provides history-based pointcuts that are defined using predicates of different expressiveness on the class graph of an object-oriented base application.

These approaches can be classified with respect to the level of expressiveness they provide, e.g., with respect to the Chomsky hierarchy or other complexity classes (see Lieberherr et al. [LPS05] for an example of the latter). Such a classification is useful because it identifies key structures that are expressible and that are not: pointcut languages of at least context-free expressiveness, for instance, can employ well-balanced structures of execution events, while finite-state based pointcut languages can not. The classification also provides an indication which kind of formal reasoning the pointcut

language can be subjected to. Approaches based on regular languages, for instance, may allow model checking techniques to be applied to analyze properties, such as aspect interaction properties, which is not possible for turing complete pointcut languages. One of the main goals of this thesis is to make a case for the usefulness of history-based pointcut languages covering the full-range of expressiveness levels, in particular, in order to improve understandability of complex programs by augmenting conciseness and declarativeness of aspects.

Advice

From a language-design viewpoint advice shares the essential characteristics of pointcuts. The advice language may be restricted to constructs of a certain level of expressivity. A finite-state based advice language is useful, for instance, in the context of the manipulation of regular protocols in order to preserve opportunities for reasoning about the resulting AO programs using model checking. The advice language may also contain constructs specific to the underlying programming paradigm, as e.g., advice for an object-oriented language that allows the manipulation of a part of the inheritance hierarchy that is relevant to a class. Finally, advice may be constructed using general-purpose or domain-specific instructions. Instructions specific to the domain of security, for example, might only allow to abort program execution in the presence of a security violation.

Aspect composition

Composition of aspects is one of the most fundamental relationship between aspects, especially because different aspects may interact. Interacting aspects should be explicitly composed in order to resolve interactions unambiguously. Depending on how fine-grained control on composition is provided, we distinguish three different cases.

Most fundamentally, aspect composition may be implicit, or explicitly defined on the language level (category Type). In most aspect systems composition is (at least partially) implicit. Composition properties may, for instance, not be expressible at the language level at all but determined by the order in which aspects are applied to the weaver. In this case, the semantics of aspect compositions may be undefined (i.e., solely determined by an essentially unknown weaver implementation), nondeterministic or equivalent to some deterministic composition scheme. In AspectJ, for instance, the composition of aspects is undefined if no precedences are declared (even if concrete weavers often implement a deterministic composition by, e.g., ordering aspect compositions according to the order of occurrence in the source code). Aspect composition may, however, be explicitly specified on the language level, in particular, using explicit precedence specifications or more general specification mechanisms.

Second, aspect compositions can be classified according to the mechanisms that are available for their definition. The most wide-spread means for explicit control of aspect composition, in particular for the resolution of interactions, is the definition of precedence or priority schemes (category Precedence). These schemes allow the definition of partial or total orderings on aspects that have to be respected by the weaver when applying aspects to a base application. Operator-based approaches (category Operator) allow aspects to be composed explicitly using aspect composition operators. Such approaches are more general than precedence-based schemes. We have defined, for instance, several instances of our EAOP model that provide such operators permitting reordering of aspects or deactivation of aspects in case of interactions (e.g., composition operators for regular aspects, see [5-DFS02b]). Finally, aspects may be composed using general composition programs (category Program). The JAsCo aspect system, for example, provides the notion of hook composition to this

end [SVJ03].

Furthermore, all aspect composition mechanisms may compose entire aspects or be restricted to advice composition (category Object). Composition mechanisms also differ with respect to how the scope can be defined during which aspect composition takes effect (category Scope). Aspect compositions may be defined invariably for all of the execution of an AO program or defined in a stateful manner that allows aspects to be combined differently over an execution depending on the current runtime state. Finally, approaches to stateful composition may be of different expressiveness (category Expressiveness): there are, for example, turing-complete ones, e.g., JAsCo's composition mechanisms, and approaches of more limited expressiveness, as, e.g.,, the operator-based approach to aspect composition we have proposed.

Aspect-specific state

Aspects may incorporate local state which can be manipulated and used as part of the other aspect abstractions, mainly pointcut and advice. This state may be general (General), that is, defined using the same mechanisms as state of the underlying base applications. In AspectJ, for instance, local aspect state comes in form of fields that can be manipulated using all available Java mechanisms. Local state may also come in more limited forms (Restricted), e.g., finite-state automata used to impose restrictions on the matching of pointcuts and application of advice.

Aspect instantiation

Aspect instantiation denotes mechanisms for the creation of new aspect instances — including allocation and initialization of fresh copies of the aspect state — depending on the occurrences of execution events of the base application or aspects themselves. Aspect instances may be tied to a large set of different execution events: creation of virtual machines in which aspects are executed (Machine), events belonging to classes (Class), creation of individual object instances (Object), matching of AspectJ-like cflow constructors (Cflow), or bindings of variables in a sequence pointcut (Binding). While most aspect languages do not provide more general instantiation mechanisms than those provided by AspectJ (which essentially consists of mechanisms of types Machine, Class, and Cflow), aspects may support very general instantiation models, such as our initial Java-based EAOP instantiation [6-DS03, 9-DS02] that supports arbitrary reflection-style aspect instantiation of type Object.

Aspect activation

Aspects may be activated only during some periods of the program execution. It is, for instance, often reasonable, to apply an aspect only after some initialization has been taken place (Init). Alternatively, activation conditions for aspects may be specified using arbitrary predicates. Note that this is related to but not the same as aspect composition and aspect instantiation. Furthermore, aspect activation can frequently be expressed simply by specific predicates to be used in pointcuts, that is, without having recourse to a full-fledged language for activation.

2.3 The atomic aspect model

We now turn to the currently predominant model of AOP, whose main representative is AspectJ. We call this model atomic because of its main characteristics, atomic pointcut languages in the sense of the taxonomy above.

The atomic aspect model is important for our investigation because it provides a de facto, very well known standard set of abstractions that has to be adopted with at most minor variations by a large number of aspect-oriented systems besides AspectJ. Such systems include aspect languages, such as AspectC++ [SGSP02], for other base languages, but also frameworks for AOP, i.e., without dedicated language support for aspects, such as frameworks for distributed components like JBoss AOP [JBoa] and Spring AOP [Spr]. Moreover, a large number of scientific investigations of crosscutting concerns, for instance Coady's work on concerns in operating systems using AspectC [CKFS01], has been conducted on the basis of an atomic aspect model. A direct consequence of this predominance of the atomic aspect model is that many users of AO technology (including some scientists) do not know about or seriously consider more expressive aspect models. Studying its characteristics and deficiencies is therefore an important element of motivation for our approach.

2.3.1 Characteristics

Atomicity of pointcuts has implications on the remaining features of the aspect model. Most importantly, the limited facilities for the directly expression of relationships between execution points on the pointcut level have to be offset by other means to record occurrences of events that influence matching of later events. This is typically achieved by manipulating aspect-internal state using advice. Moreover, atomic pointcuts impede the precise but flexible definition of how to compose aspects: mainstream aspect languages therefore only provide very limited support for aspect composition. We now consider these characteristics in detail.

Atomic pointcuts

Aspect languages for object-oriented base languages typically contain a rich set of basic pointcut constructors and a small set of operators over pointcuts, in particular, logical operators. Basic pointcut constructors typically belong to the following four categories:

- *Calls and fields:* Such constructors match calls, related execution events (such as object initializations and exception handlers), as well as field set and get operations.
- *Class:* constructors matching on the static structure of classes or manipulating them (e.g., by modifying the class hierarchy or by introducing new interfaces and class fields).
- *Control flow:* constructors allowing execution events to be matched that occur from a given call to the corresponding return or a subregion thereof.
- *General predicate:* constructors that allow to make depend pointcuts on general conditions.

AspectJ includes such a set of basic pointcut constructors: matching of calls can distinguish between caller and callee sides by respectively using the `call` and `execution` constructors, calls may be restricted to occur within methods statically defined within a class, "intertype declarations" allow to introduce new fields in class definitions or to modify the class hierarchy, control-flow based matching may exclude the originating call (`cflowbelow`), and the `if`-constructor allows arbitrary Java predicates to be used in pointcuts.

Two of the four constructor categories — calls/fields constructors and control flow constructors — designate execution events. The other two categories can be interpreted as predicates defining static or, in the case of the `if`-constructor, dynamic scopes. (We do not consider intertype declarations, because their modifications of the static structure of a program are not relevant to arbitrary base paradigms.)

All call and field constructors denote individual events, i.e., they are atomic pointcut constructors. Control flow constructors are non-atomic— in fact, the only non-atomic constructors in AspectJ — in that they relate all execution events between a call event and the corresponding return event. However, all of these pointcut constructors provide only very limited means to construct pointcuts that match subsequences of the execution history.

Furthermore, the few operators that can be commonly applied to pointcut expressions do not improve on this state-of-affairs. For instance, logical operations on pointcuts built using atomic pointcut do not allow to express non-atomic relationships.

Aspect-specific state

In order to offset this lack of expressiveness, a general notion of state and corresponding manipulation mechanisms are typically used. Three kinds of mechanism are common. First, aspects may define internal state that can be used to record information about previous execution events and used to restrict the matching of or provide additional information for later events. A well-known example of the latter are memoization aspects. Second, aspects may access state of the base application and, in most systems, also modify it. This can often be done implicitly through some representation of the state at the execution point a pointcut matches; in AspectJ, for instance, by a reflective access using the `thisJoinPoint` variable. Third, the state of the base program can often be manipulated using static mechanisms, most notably, by using aspects to add state variables to classes of the base application.

Turing-complete advice

In order to manipulate advice-internal state and the state of the base application, a powerful advice language is necessary. Commonly, advice is implemented using a turing-complete language that frequently results from enriching the base language by a small set of aspect-specific mechanisms. In AspectJ, for instance, the advice-specific pseudo-method `proceed` is used to call matched base functionality and the `thisJoinPoint` mechanism for reflective access to contextual information.

Limited composition mechanisms

The atomic aspect model provides only very limited means to handle the composition of aspects. In terms of the taxonomy summarized in Fig. 2.1, aspect composition is typically left implicit, that is, not explicitly controlled on the language level or governed using explicit precedence specifications. AspectJ provides another mechanism that belongs to the category of programmatic composition features: pointcuts may test whether an execution event occurs in the scope of an advice that is defined in other aspects by means of the `adviceexecution` constructor.

None of these three mechanisms comes close to a general notion of operator-based or programmatic aspect composition. Defining composition implicitly yields at best difficult-to-understand composition semantics, prioritization often is too inflexible, and the available programmatic means in AspectJ are too fine-grained. Note that all of these problems are (at least partially) grounded in the lack of expressive means to define pointcuts relating execution points: implicit composition and precedence specifications do not take into account individual execution states but compose aspects uniformly over the entire execution of an application; to the contrary, fine-grained advice execution control does not provide declarative control over sequences of execution events.

2.3.2 Drawbacks

Because of these characteristics the atomic aspect model is subject to two important problems. First, crosscutting concerns that can only be modularized in terms of traces of the execution history can be described only cumbersomely. Second, the correctness of AO programs developed in this style is often very difficult to judge and almost always impossible to ascertain formally. In this section, we briefly review these two problems.

Lack of conciseness of aspect definitions

As motivated in Sec. 2.1 many crosscutting concerns require different events occurring during an execution trace to be accounted for in order to modularize them. Such concerns cannot be concisely modularized using atomic aspect languages because of a lack of sufficiently expressive mechanisms in the pointcut language. Hence, these concerns can be implemented only by breaking down the dependencies into atomic pointcuts. In particular, it is very frequently not possible to use control-flow pointcut constructors, because the calls corresponding to later relevant execution events are not nested in the control flow of the previous ones. Concretely, the corresponding implementations of trace-based crosscutting relationships are therefore structured in three parts using atomic aspect languages:

- An atomic pointcut definition is defined for each relevant execution point.
- A suitable state is set up that allows to record all information of previous execution points that are required at later points. This concern-specific state commonly involves dedicated aspect-internal state but frequently reuses state already part of the base application.
- Each pointcut definition triggers advice that updates the concern-specific state and applies the necessary modifications to the base application.

The resulting implementations are characterized by a large number of separate but implicitly related pointcut and advice definitions. Frequently, the pointcuts and advice simulate higher-level abstractions, such as finite-state automata, that could be used to express them explicitly using more expressive aspect systems. Both issues obviously lead to AO programs that lack concision and therefore are difficult to understand.

Program correctness

AOP generally poses a hard correctness problem because traditional reasoning mechanisms that have been developed for non-crosscutting programming structures are not applicable to AO programs. Traditional reasoning mechanisms rely, in fact, on composition and encapsulation properties that do not hold for modifications on an application induced by aspects. Consequently, informal justification of the correctness of AO programs as well as their formal, possibly automatic, validation and verification remains a mostly open issue. While we consider the general problem later in Chapter 3, it is important for the present discussion to note that the atomic aspect model exacerbates the aspect correctness problem to a large extent.

The most obvious handle to express and ensure properties about crosscutting concerns are properties of pointcuts. However, the limited expressiveness of pointcuts in the atomic aspect model only allows very simple and coarse-grained properties to be captured this way. A notable example addressing properties on the execution of AO programs is the work by Kiczales and Mezini [KM05], which permits to determine coarse-grained upper bounds on which parts of a base application are affected

by aspects and to ensure certain control flow properties. Such properties would, however, be of limited use in case of the trace-based relationships discussed above. A second notable group of work investigates the effects of intertype declarations in AspectJ-like languages. Wilke et al. [HNBA06], for instance, present an algorithm that can be used to detect ambiguous declarations, i.e., that result in non-deterministic aspect weaving.

Another problem for the evaluation of the correctness of atomic AO programs is the use of aspect-specific state and turing-complete advice languages to handle that state: basically aspect-internal state constitutes a global state with respect to advice and application-level state is even global with respect to different aspects. This means that the formulation of correctness properties, their analysis and verification is subject to all problems arising from globally shared state that are already present in the base paradigm in addition to the correctness issues originating from the aspect-specific abstractions.

Finally, the atomic aspect model also impedes expression and verification of properties because of its limited means for aspect composition. This is particularly problematic because it almost entirely inhibits reasoning about interactions of aspects. Consequently, there are only very few approaches to tackle this very important problem and the corresponding results are typically very imprecise. Furthermore, most aspect programmers currently only use very coarse-grained techniques to identify interactions that are often provided on the level of development environments: the AspectJ Eclipse plugin, for example, marks different advice that may match the same method call.

2.4 Event-based AOP: beyond atomic pointcuts

We are now ready to introduce a general aspect model, the model of Event-based Aspect-Oriented Programming (EAOP)[4] that Rémi Douence and me have started developing in 2000 in order to address the issues of the atomic model introduced above.

2.4.1 Characteristics

The EAOP model has been developed as an aspect model having the following main characteristics:

- Language support for expressive aspects
- Language support for explicit aspect composition
- Support for reasoning about properties of AO programs
- Integration with mainstream base languages
- Reasonably efficient implementation

The first three directly address the drawbacks of the atomic aspect model discussed in the previous section. The fourth is motivated by the observation that many approaches for expressive aspect languages are based on other programming paradigms, such as logic languages. Such approaches, while providing rich facilities to express relationships, cannot simply be used with object-oriented and imperative languages. The last is useful because some interesting expressive aspect languages are inherently difficult to implement efficiently, e.g., Masuhara et al's data-flow pointcuts [MK03a].

[4] Note that, strictly speaking, EAOP is a metamodel in the sense that it defines a set of general characteristics that we have then instantiated by different concrete aspect languages. Since the references to the EAOP metamodel and the instantiations cannot give rise to confusion in the present text, we refer to EAOP simply as an "aspect model".

Non-atomic pointcuts

The main motivation for our work has been the need for aspect languages allowing to represent relationships between execution events of an application. Many different formalisms and programming paradigms may serve as a basis for such an endeavor, for instance logics such as Prolog and temporal logics, functional calculi, grammars and automata. Today, most of these approaches have been and are under active investigation but, when we started our work in 1999 on what we later called the EAOP model, only the AspectJ model was known.

After a first approach defining pointcuts in a functional setting [5-DMS01a], we have decided to focus on pointcut languages that are based on different forms of *automata*, in particular, finite-state automata, counter automata, and visibly pushdown automata (VPA) [AM04].

This choice is justified by four different advisements. First, the most basic (and practically relevant) relationship between execution events that extends the atomic pointcut model (including control flow-relationships) are pointcuts over sequences of events: these form a natural special case of automata-based pointcut languages. Second, automata-based pointcuts mostly have quite *intuitive interpretations* (e.g., finite-state automata as regular expressions and regular grammars, VPA-based structures in terms of well-balanced contexts) and the resulting pointcut expressions therefore are easily understandable. Third, many automata-based pointcut languages do not provide for turing-complete relationships: they thus enable specialized, effective and, partially, efficient formal methods to be used to reason over pointcut expressions. Fourth, the runtime matching of automata-based expressions is supported by a large body of work treating their optimized implementation.

Explicit support for aspect composition

Our work has been guided by the desire to provide powerful and flexible means for aspect composition on the language level, in particular improving on the drawbacks of the atomic aspect model in this respect. Concretely, we have strived to meet the three following characteristics. First, aspect composition should be *explicit*: the model should support composition specifications over aspects as entities of their own. Second, aspect composition should be *stateful*, that is, flexibly composable in terms of the runtime state of an AO application, in particular, the states of pointcut expressions. Stateful aspect composition subsumes support for the composition mechanisms of atomic aspects (in particular, precedence schemes, AspectJ's `adviceexecution`) but also includes operators that compose aspects differently over different parts of the execution of an AO program. Third, we are interested in investigating composition of aspects, aka. aspects of aspects: the need to handle the effects of aspects on other aspects rapidly arises if aspects are used as a general means for constructing applications.

Support for reasoning about aspects

The key characteristic that differentiates EAOP from atomic aspect models, non-atomic pointcuts, is instrumental to informally establish properties of aspects and next to indispensable in order to establish them formally. Important properties of aspects that should be addressed by a model for AOSD include interaction properties of non-independent aspects, correctness properties of specific aspects and optimization properties enabling the efficient implementation of AO programs.

Limiting the expressiveness of pointcut expressions allows programmers to grasp the meaning of aspect definitions much easier. Furthermore, defining applications using specialized well-known classes of languages is a well-known means to support better understanding of programs by developers and users: finite-state machines, for instance, have been used in many application domains to this

end. Limited expressiveness also much augments the opportunities for formal and, in particular automated reasoning about properties, compared to the atomic pointcut model. First, structures based on finite-state automata or visibly pushdown automata allow standard operations, such as product operations, to be used for the analysis of aspect programs. Furthermore, suitable abstractions can be used to automatize formal verification tasks efficiently, for instance, through the use of model checking techniques.

Integration with mainstream languages

Relationships between execution events relevant for aspects can be made explicit using fundamentally different formalisms and supporting techniques. As indicated before different logics, calculi, grammar-based approaches, etc., have been proposed to this end by now. Whether an aspect system can be integrated smoothly with an underlying base programming paradigm depends, however, on the combination of aspect language and base paradigm at hand: the integration of general logic programming languages and object-oriented runtime systems, for instance, is still an active open research issue.

To the contrary, the automata-based aspect structures we have been mainly worked on are particularly well-suited for several reasons for the integration of aspect languages with object-oriented and imperative base applications. First, implementations of automata in form of libraries and OO frameworks can be used to implement automata-based aspect systems, that is, special syntax extensions are not necessarily needed. Second, modifications to the base application may in general have repercussions on the state that governs the matching of pointcuts. In automata-based approaches such modifications can be readily expressed in terms of modifications to automata states. Finally, automata-based approaches are also appealing from a pragmatic point of view because of the many existing software implementation of the base operations, model checkers, etc., that can be reused in an AO context.

Reasonably efficient implementation

AOSD is a relatively recent research domain and, as such, efficiency of implementations has mostly been an issue of minor importance. While this general consideration is still correct to some extent, efficiency is important and sometimes crucial to various of its applications. Middleware and systems software, in particular, constitute two large application domains of this kind (see [2-CJS06] for a more detailed discussion).

In a model with non-atomic pointcuts, the complexity of the pointcut language and its implementation is of foremost importance as far as efficiency of implementation is concerned. Pointcut languages may easily include features which by principle can not be implemented with reasonable efficiency, for instance, if the pointcut language relies on full-fledged logical inference or general dataflow relationships (as, e.g., in Masuhara et al's approach [MK03a]). Automata-based aspect structures are very attractive in this respect because their overhead (i) is determined by well-known language properties, (ii) can be kept small to very small with reasonable effort, and (iii) can typically be evaluated well by programmers.

Besides the pointcut language, advice may also cause AO-specific overhead. One issue is whether advice may generate new execution events that are subject to weaving: in this case, repeated execution of event generation by advice followed by pointcut matching and reapplication of advice may result in resource-intensive weaving. Most aspect models do not support such general weaving strategies, which are akin to powerful intercession mechanisms in systems using computational reflection. As-

pectJ, for instance, is designed to allow weaving through compile-time code transformation, which simplifies the task of keeping weaving efficient. The EAOP model has been kept generic enough to allow the investigation of such alternative weaving strategies (We will come back to the issue of visibility of advice to weaving and corresponding weaving strategies in Chapter 3.).

Finally, the remaining features of aspect models introduced in Sec. 2.3.1 also influence efficiency. Aspect instantiation, in particular, may be problematic if aspect instances can be generated at runtime. Per object instances of aspects that are instantiated at runtime, for instance, incur a non-negligible overhead. Furthermore, manipulation of aspect-specific state may be costly, for instance, if such state may be shared in a distributed environment. The composition of aspects can also cause execution overhead, especially if it is performed at runtime. Once again, in contrast to more restricted models, the genericity of the EAOP model allows exploration of these issues as presented in the following chapters.

2.5 Conclusion and perspectives

The main motivation of our work on expressive aspect models has been the observation that many interesting crosscutting concerns can be defined only in terms of complex relationship between execution events and cannot be modularized well using atomic aspect systems. Such systems fall short to provide concise aspect definitions and the correctness of aspects cannot be formally established. The EAOP model allows to tackle these shortcomings by making explicit such dependencies, yielding better structured and formally defined aspect languages that are amenable to manual or automatic property verification.

Perspectives. The results reported on in this section pave the way to address a number of other important open problems.

Stateful pointcut and aspect languages are a tool of much wider applicability than explored by us. Our notion of regular stateful aspects has been integrated by the SSEL group at Vrije Universiteit Brussel into the aspect system JAsCo [VSCDF05] and applied, among others to business rules and web services. In a subsequent cooperation, we have applied stateful aspects for web services to distributed web service compositions [5-BNSVV06].

We have underlined the facilities for aspect composition provided by the EAOP model. However, composition properties of aspects are not limited to the compositional construction of aspects or composition of aspects with one another but also raise the question of how to integrate aspects with existing composition mechanisms, such as software components and modules. Much research is currently under way on this issue. We come back to the perspectives opened up by our work in Sec. 6.

Chapter 3

Foundations of AOP

We now turn to the fundamental problem of how AOP can be given a rigorous foundation. While semantics of programming languages as well as analysis, verification and enforcement of their properties is a well-established field in the domain of traditional programming paradigms, AOP requires new formalisms and techniques to be developed for the rigorous treatment of crosscutting relationships. Work on the formal treatment of AO programs has only started at the end of the nineties and therefore still is rather immature, essentially applying existing formalisms to properties of AO languages. Concretely, this work focuses on formalisms and techniques (i) to define the semantics of AO programs and (ii) to determine properties of aspects and support enforcement of properties over aspects.

To cite a few examples, this exploration of existing formal frameworks for the definition of the semantics of aspect languages has led to the development of different operational semantics [Läm02, 5-DFS02b], denotational semantics [WKD04] and process calculi [And01, BJJR04] for aspects. Furthermore, a variety of properties of AO programs have been investigated, e.g., safety properties of aspects [DW06], yielding in particular different classifications of aspects with respect to their semantic effects on the base program [Kat06]. Finally, the automatic analysis of properties using, for instance, static analysis techniques [DW06] and model checking [KS03] has been considered.

In this chapter we present our work on the formal semantics and support for properties of so-called *stateful aspects*, a term we have introduced in [5-DFS02b] to designate languages with explicit support for aspects defined in terms of the execution history of an AO program. (This term has been taken over by other researchers since then, see, e.g., [VSCDF05, VJ05].) Concretely, the results presented here address three different categories of problems:

- *Semantics of aspect languages*. We briefly motivate and describe two different semantic frameworks — a functional and an operational one — we have developed as one of the first semantics for AOP.
- *Properties of aspects*. We present static analysis techniques we have developed for two of the most fundamental properties of AO programs: interaction among aspects and the correct application of aspects to base programs.
- *Aspect composition.* We also detail means for the rigorous treatment of aspect composition issues, in particular based on stateful aspect composition operators.

We concentrate here on mechanisms for the semantics of aspects that abstract from specific aspect languages and specific uses of aspects. As an example of semantics for specific aspect languages,

two formal transformational semantics of aspects for C programs are discussed in Sec. 4.2.2. We do, however, not present our work on the formalization of aspects and aspect weaving through program transformation [7-FS98, 7-FS99] that Pascal Fradet and me have started with and also do not present the work using the temporal logic CTL for the formalization of pointcuts in terms of sets of paths over the control flow graph of C-programs [5-ÅLS+03].

The work presented in this chapter has been done in the context of different collaborations. Most of the work on the semantics for and properties of stateful aspects has been done with my colleague Rémi Douence and Pascal Fradet from INRIA. The recent work on VPA-based aspects has been started as part of Ha Nguyen's PhD thesis that I am currently supervising.

The remainder of this chapter is structured as follows. Section 3.1 presents three different semantics for stateful aspects of different expressiveness by extending the uniform description framework we have introduced in [1-DFS04b]. In Section 3.2, we consider how interaction properties and applicability conditions of AO programs can be defined and analyzed using stateful aspects. A formal approach to aspect composition is set forth in Section 3.3.

3.1 Formal semantics for stateful aspects

The core of our work on formal semantics for aspect languages has been motivated by two key observations. First, the absence of any formal foundation for aspect languages when we started our studies in this domain in 1997. Second, most aspect languages — and hence corresponding formal frameworks — have been instances of the atomic aspect model as introduced in the previous chapter. The resulting poor representation of the semantic relationships that govern aspects has set off our quest for more expressive semantic frameworks that make explicit such relationships or at least support their analysis.

Our response to this challenge consisted in a series of different but related semantic frameworks for the formalization of aspects defined over the history (that is, traces) of the execution of AO programs. Furthermore, the expressiveness of the aspect languages of these frameworks has been chosen so that properties over aspects can be formally established.

3.1.1 Presentation framework

We first turn to the problem to giving precise semantics to aspect languages. In the following, we give a unifying presentation of three semantic frameworks for stateful aspects we have developed:

- Aspects whose pointcuts are defined using a functional calculus [5-DMS01a, 7-DMS01b].
- Finite-state based, i.e. regular, aspects [5-DFS02b, 9-DFS02a, 5-DFS04a].
- Visibly pushdown automata (VPA) based aspects [NS06].

These three approaches cover a large part of the aspect language design space. First, the three aspect languages are of different expressiveness ranging from the relatively limited regular aspects, via VPA-based aspects that allow a form of well-balanced contexts to be expressed and manipulated, to the turing-complete pointcuts defined using a functional calculus. Second, they have been defined using different semantic frameworks: an equational theory in the case of aspects with functional pointcuts, regular aspects and VPA-based aspects by means of a small-step operational semantics, the latter of which defined after a translation into plain VPAs. Third, the three languages admit different

techniques to be used for property verification: equational reasoning about functional pointcuts, static analysis in the case of regular and VPA-based aspects.

The main characteristics of these language can concisely be presented based on the following general definition for stateful languages (which is a slight extension of that defined in [1-DFS04b]):

$$\begin{array}{rcll} A & ::= & P \rhd I \mid P & \textit{; basic aspect} \\ & \mid & A_1; A_2 & \textit{; sequence} \\ & \mid & A_1 \,\square\, A_2 & \textit{; choice} \\ & \mid & \mu a.A & \textit{; recursive definition} \\ & \mid & a & \textit{; recursive call} \end{array} \tag{3.1}$$

This grammar allows the definition of aspects A that are constructed from basic aspects using three aspect composition operators: sequencing '; ', choice '$\square$', and repetition '$\mu a \ldots a$'. Basic aspects define pointcuts P that may trigger inserts, that is advice, I; empty inserts may be omitted.

The generality of the terms admitted by this grammar, in particular compared to the finite-state aspects considered later, results from recursive calls can occur at arbitrary places in terms. The corresponding class of languages accommodates a high degree of variability by (i) different concrete definitions for pointcuts and advice (e.g., to introduce composition operators in the advice sublanguage I), (ii) different semantics for the basic operations (e.g., deterministic vs. non-deterministic choice) and (iii) structural restrictions (e.g., restricting recursive definitions to tail recursions).

In the following, this formal framework is extended compared to the presentation in [1-DFS04b] in two respects. First, we show how to include means for aspects based on visibly pushdown automata besides turing-complete and regular aspects. Second, we accommodate different mechanisms for the composition of aspects on the pointcut and advice level.

3.1.2 Aspects with functional pointcuts

One of the reasons of the popularity of atomic aspect models, such as AspectJ, stems from the fact that they typically allow to easily define arbitrary pointcuts. As discussed in the previous chapter, complex pointcuts have then to be implemented using atomic pointcuts and pointcut-external state, thus impairing understandability of aspect definitions and essentially foreclosing reasoning about pointcuts. A solution to this trade-off between generality and tractability can be developed from an analogous situation in traditional programming: imperative programming also suffers from similar tractability problems. Alternative programming paradigms, such as functional programming and logic programming, have been developed as a remedy. The trade-off induced by atomic pointcuts languages can be similarly addressed. To this end, we have explored the definition of pointcuts as functional expressions that are defined in terms of monadic parser expressions [HM98].

Language. We have considered pointcuts defined as values over a recursive datatype that include, inter alia, constructors for the matching (and extraction of information) of individual joinpoints, a sequence constructor and a parallel constructor that realizes a form of disjunction. Specific binding constructors may be used to pass values extracted from one joinpoint to other parts of a pointcut definition.

The essentials of such a language can be easily represented in terms of the Grammar 3.1: recursive definitions, sequencing, and the parallelization operator are represented by the operators μ, ; , $\square$, respectively. Individual joinpoints may be represented by recursive constructor terms involving variables in argument positions, the latter to extract data from joinpoints for later use. Note that we abstract

here from the specifics of joinpoint matching, e.g., return types and wildcard matching. Furthermore, pointcuts may test for arbitrary conditions using an *if*-constructor. We represent value binding by distinguishing between assignment of variables, noted $\bar{v}$ for variable v from variable uses.

> **Example:** An aspect for updating information on files and trust of their respective providers as part of an P2P application can be defined as follows:
>
> $$\mu a.\bar{f} = \mathit{localCache};\ (\mathit{if}(f \neq \mathit{null}); a \\ \Box\ \mathit{if}(f = \mathit{null}); (\mu b.\mathit{files\_query}; b; \mathit{files\_reply} \triangleright \mathit{updateFileInfo}; a \\ \Box\ \mu c.\mathit{trust\_query}; c; \mathit{trust\_reply} \triangleright \mathit{updateTrustInfo}; a))$$
>
> This aspect first looks up a file in a local cache and, if it has not been found there, updates the local cache after a recursive file query or a recursive trust query.

Semantics. An aspect language providing expressive pointcuts can be defined in terms of a monitor that observes the base program execution, matches pointcuts and triggers advice execution when appropriate. Functional pointcuts, which form a sublanguage of pure (a.k.a. side-effect free) functional programming languages [Hud89], are particular attractive in this context for three reasons. First, pure functional languages come equipped with formally-defined and relatively simple semantics. Hence, functional pointcuts can be formally defined by translating them into plain functional programs. Second, the corresponding interpreters can be implemented directly in terms of functional programming languages. Third, functional pointcuts are referentially transparent [Hud89], which enables equational reasoning to be used to establish formal properties over pointcuts. Furthermore, this property carries over to aspects built using functional pointcuts as long as the aspect does not rely on and modify (aspect internal or base program) state through advice.

We have harnessed this method by defining a translation of functional pointcuts into the pure functional language Haskell [Has]: recursive composition is translated into recursive functions, sequences into nested function applications and the two branches of a choice operation are explored (on the semantic level) in parallel until the first match is encountered, the decision being non-deterministic in case both match at the same point. (The reader might have reckoned that a choice has to be resolved based on the first event encountered after the choice point: this is a reasonable alternative semantics that we have explored as part of the regular aspects discussed in Sec. 3.1.3.) Concretely, this translation has been realized in form of an equational system relating different constructor terms, which includes the following three laws

$$\begin{aligned} \mathit{filter}(p)(\mathit{return}(e) \Box p_2) &= \mathit{return}(e) \Box \mathit{filter}(p)(p_2) \\ p_1; (p_2 \Box p_3) &= (p_1; p_2) \Box (p_1; p_3) \\ (p_1; p_3) \Box (p_2; p_3) &= (p_1; p_2) \Box p_3 \end{aligned}$$

stating, respectively, that filtering on branches continues as long as a branch has not yet been fully explored (i.e., until a match is returned) and that sequencing distributes over choices from the left and from the right.

This equation system induces a rewriting system that has been used to formally define and implement a monitor-based weaver for functional pointcuts.

Aspect properties and equational reasoning. As mentioned above, functional pointcuts are amenable to equational reasoning because they are referentially transparent. This allows common equivalences over functional programs, including, for example, inductive reasoning over recursive equations, to be used to prove properties over aspects. Furthermore, equational reasoning over functional pointcuts is supported by the equational definition of the basic pointcut constructors. We have shown [5-DMS01a], for instance, how inductive reasoning and the application of the laws linking sequences and choices can be used to prove the equivalence of an AspectJ-like *cflow* pointcut constructor using a runtime stack to a constructor not needing a stack, an optimization that has been the subject (in more general settings) of a number of later publications by other researchers [MKD02, ACH+05]. Since functional programming languages are turing complete, these proofs can in general not be automatized, but they are facilitated by the large number of existing properties over functional programs, especially the so-called theory of lists [Bir89].

3.1.3 Regular aspects

While functional pointcuts support the formal definition and reasoning about AO programs, equational reasoning over turing-complete expressions cannot be automatized. However, it is well known that the automatic or semi-automatic analysis of program properties is useful to handle correctness issues especially of large scale programs (to which application of manual techniques simply is too unwieldy). Restricting the expressiveness of the aspect language therefore seems highly worthwhile if it bolsters automation. The class of languages with the most substantial automatic support for analysis of correctness properties are regular languages, i.e., languages that can be defined in terms of finite-state automata. This language class especially supports correctness properties tackled using static analysis techniques, such as abstract interpretation [JN94] and model checking [CGP99].

A notion of regular aspects, which constrains the semantics of aspects using finite-state automata, therefore seems to be a natural means to leverage existing analysis techniques. Furthermore, regular aspects are still expressive enough to capture a large number of interesting non-atomic relationships between joinpoints, see, e.g., Allan et al. [AAC+05] for a series of examples from the domain of program development.

However, there are several non-trivial issues in the definition of regular aspects. A first question is if regularity is only required of pointcuts or should express constraints over pointcuts and advice. Regular pointcuts could be defined e.g., by restricting P to regular expressions in the language of Grammar 3.1. Allan et al. [AAC+05] have introduced "tracematches" that also define pointcuts as regular expressions. Regular pointcuts raise the problem that they do not allow to define and analyze properties of complete aspects: the main question in this context being how to take into account the effects of advice. If advice can generate joinpoints of its own, i.e., advice is visible to aspect weaving, the behavior of the aspect cannot be simply defined in terms of the composition of regular pointcuts. We have therefore opted to define a notion of regular aspects where regularity constrains the execution of complete aspects and to address the problem of the visibility of advice explicitly.

A second issue is how to integrate dependencies introduced by variables that are assigned and used as part of matching in regular pointcuts. Depending on the scope of variable definitions, regularity can be violated: a dynamic notion of scope which allows to shadow variables of the same name assigned in a later iterations of a repetitive definition, for instance, cannot be represented by regular expressions. Allan et al. [AAC+05] impose suitable restrictions by requiring variables to be bound consistently to the same value over all occurrences. Our solution presented below defines a notion of dynamic scope where a variable binding extends from one binding to the next binding occurring in an execution trace (or the end of the term representing the current aspect).

We are now ready to present our specialization of Grammar 3.1 for regular aspects:

$$\begin{array}{lcll} P & ::= & \mathtt{f}\ T_1 \ldots T_n & \textit{; (atomic) pointcuts} \\ T & ::= & \mathtt{f}\ T_1 \ldots T_n \mid \bar{x} \mid x & \\ A & ::= & \mu a.A & \textit{; recursive definition} \\ & \mid & P \rhd I; A \mid P; A & \textit{; prefixing} \\ & \mid & P \rhd I; a \mid P; a & \textit{; end of sequence} \\ & \mid & A_1 \,\Box\, A_2 & \textit{; choice} \end{array} \tag{3.2}$$

where the set of term variables x and the set of recursion variables a is to be disjoint. Pointcuts P are nested constructor terms, that may assign variables, noted $\bar{x}$, through matching and use them, noted x, to restrict matching. Aspects A are restricted compared to Grammar 3.1 in that recursive definitions may not end in the middle of sequences: this ensures tail-recursiveness of regular aspects and thus equivalence to regular expressions.

Note that this language is a simplified version compared to our previous work: it does not include pointcuts constructed from logical operators (as [5-DFS04a]) does) and in contrast to [Far03, NS06] only a restricted set of regular pointcuts is included (in form of variants of prefixing and sequences with omitted advice, i.e., *skip* advice). Both of these features are not essential to the discussion here, since they do not interfere with the formal definition of a semantics for such a language and analysis of aspect-specific properties.

> **Example 3.1.1:** Reconsider the example of file queries in the P2P domain, we now cannot express aspects about recursive queries anymore, because regular structures do not allow to correctly match nested structures. Regular aspects can, however, still express many useful relationships between joinpoints, e.g., abortable file queries over one channel that update file information on the current node only in case the query is completed:
>
> $$\begin{array}{l} \mu a.\bar{f} = \mathit{localCache};\ (!\mathit{isNull}(f); a \\ \qquad\qquad \Box\ \mathit{isNull}(f); \mu b.\mathit{files\_query}; (\mathit{files\_reply} \rhd \mathit{updateFileInfo}; b \\ \qquad\qquad\qquad\qquad \Box\ \mathit{abort}; a)) \end{array}$$

Semantics. The semantics of functional pointcuts has been defined in terms of a translation into the functional programming language Haskell in order to harness the large body of results of the equational theory over pure functional languages. However, in order to investigate fundamental properties of aspect languages, such as interaction among aspects and different weaving models, a lower-level semantics, which allows to define and reason about arbitrarily fine-grained evaluation steps of an AO program seems better suited. By now a fair number of different kinds of semantics have been explored as part of this endeavor, for instance, process calculi by Andrews [And01], denotational semantics for an AspectJ-like language by Wand et al. [WKD04], semantics for regular pointcuts by Alan et al. [AAC+05], and operational semantics by Walker et al. [WZL03] as well as Clifton and Leavens [CL06].

Preceding almost all of this work, we have equipped regular aspects with the first formal semantics in terms of a small-step operational semantics [5-DFS02b]. This semantics precisely defines three issues:

1. The modification of base program executions through aspect weaving.

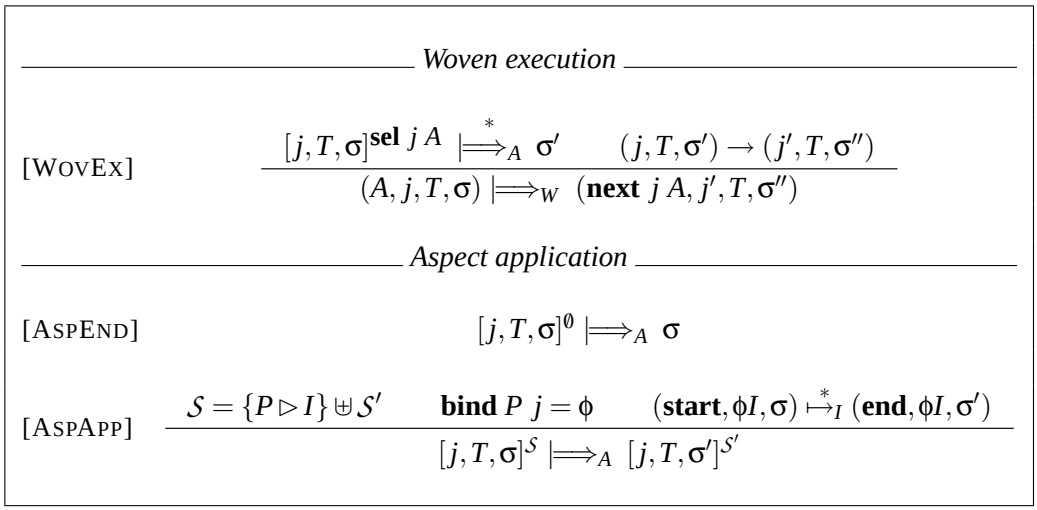


Figure 3.1: Weaving of regular aspects

2. Weaving of several aspects at a time.

3. Visibility of advice to other aspects, i.e., whether advice is subject to weaving.

In the following we will present the essentials of how these three different issues are handled in a formal way (see [9-DFS02a] for a detailed presentation including examples).

Modification of base executions through aspect weaving. Aspect weaving is defined as shown in Fig. 3.1. The execution of the base program as well as the woven program is abstracted into a relation defining execution steps that transform runtime configurations. A configuration (j,T,σ) consists of the current joinpoint j, the static program text T and the dynamic state σ.

The weaver is essentially defined in terms of two transition relations. The woven execution $\Longrightarrow_W$, see inference rule WovEx, yields a follow aspect, next joinpoint and new state after weaving a set of aspects at the current joint point, as expressed by the aspect application relation $\Longrightarrow_A$, and advancing the base execution to the next join point (base transition relation $\rightarrow$). Second, weaving of one basic aspect $P \rhd I$, see inference rule AspApp, yields a new dynamic state by calculating a variable binding ϕ resulting from matching the pointcut P on the current joinpoint j and executing ϕI, the advice after substitution by the generated variable binding.

Multiple aspects. The aspect selection function **sel** used in rule WovEx is defined over a set A of aspects and yields all basic aspects that match the current joinpoint. The aspect application rule AspAdd then iterates over all these aspects. Once all aspects have been applied, rule AspEnc, returns the resulting new dynamic state.

Note that the rule AspAdd non-deterministically chooses one of the applicable basic rules. The weaver definition could obviously be modified to support deterministic orderings, as has been done in a number of approaches, e.g., Andrew's aspect calculus [And01]. However, our approach is more general and flexible: interactions between aspects, i.e., the applicability of several aspects at a joinpoint, can be analyzed statically (see Sec. 3.2) and the resulting conflicts be resolved using aspect composition operators, amongst others, by ordering them. We therefore generalize on most existing

approaches, e.g., AspectJ, that support only partial orderings of aspects that are applied simultaneously. This is discussed in detail in Sec. 3.3.

Visibility of aspects to other aspects. Visibility of aspects refers to the property if the functionality introduced by aspects, i.e., advice, is subject to weaving itself, in which case other aspects can be applied to this functionality. The weaver definition of Fig. 3.1 accommodates different models of visibility simply by choosing appropriate transition relations for the execution of inserts, i.e., the relation $\mapsto_I$ in rule ASPADD: if this relation is chosen to be the base transition relation $\rightarrow$, advice is not woven and thus not visible to other aspects; if the woven execution relation $\mid\!\Longrightarrow_W$ is chosen, advice is woven and thus visible to other aspects.

3.1.4 VPA-based aspects

Aspects relying on functional pointcuts (or more generally on turing-complete pointcuts) and regular aspects delimit a potentially large set of aspect languages that differ with respect to expressiveness. The turing-complete ones are more interesting because of the larger set of crosscutting relationships they allow to express, regular aspects support fewer relationships to be captured but enable automatic support for properties analysis over aspects.

Researchers in the field of formal languages have defined a variety of classes of languages whose expressiveness lies between regular and turing-complete classes, for example, LL/LR-languages, context-free and context-sensitive languages. All of these classes have their *raison d'être* and should, in principle, be useful to describe crosscutting relationships. However, until now only very few work has been done on aspect languages belonging to these classes. Notable exceptions are the notion of context-free aspects proposed by Walker and Viggers [WV04] and our suggestion of aspects over context-sensitive protocols [5-Süd05]. While worthwhile on the grounds of their expressiveness, these approaches are subject to the problem that the underlying language classes are not readily amenable to automatic analysis and reasoning techniques, difficult to implement or both.

There is substantial work currently on formal languages (not aspect languages) that are more expressive than regular languages but that improve with respect to automatic reasoning and implementation support compared to context-free languages. Recently, *visibly pushdown automata* (VPA) [AM04] have been introduced as a means to define a language class, visibly pushdown languages (VPLs), that is (i) strictly more expressive than regular ones, (i) strictly less than context-free languages and (iii) that obeys all common closure properties of regular languages. The third characteristics distinguishes visibly pushdown languages from context-free ones, which are, for instance, not closed under the union operation. VPLs thus support a much larger class of (automatic) analyses than context-free ones. Technically, this is achieved by trading off some expressivity of VPAs compared to plain pushdown automata: the set of transition labels in VPAs must be partitioned in those that push data on the stack, pop from the stack or do not modify the stack.

Based on this observation, we have introduced VPA-based aspects, whose essentials can be represented as an extension to regular aspects by the following grammar:

$$
\begin{array}{lcl}
P & ::= & \mathtt{F}\ T_1\ \ldots\ T_n \\
F & ::= & f \mid f_s \mid \underline{f_s} \\
T & ::= & \mathtt{f}\ T_1 \ldots T_n \mid \overline{x} \mid x \\
A & ::= & \mu a.A \mid P \rhd I; A \mid P; A \mid P \rhd I; a \mid P; a \mid A_1 \,\square\, A_2 \\
 & & \textit{; same as regular aspects (see Grammar 3.2)}
\end{array}
\tag{3.3}
$$

Here, VPA-based aspects extend the term structure defined by Grammar 3.1 while preserving the structure of regular aspects A. Pointcut terms are modified in two ways. Most importantly, there are now three sets of term constructors:

- Constructors that match functions (or methods) f as in the case of regular aspects.
- Constructors f_s that match f and put s on the call stack.
- Constructors $\underline{f_s}$ that match f only if a corresponding call has occurred, i.e., s can be popped from the call stack.

A second difference is that we allow stack manipulating calls only on the outermost level of nested pointcuts terms[1].

> **Example:** The following example shows how VPA-based aspects can be used to modify protocols involving recursive file queries:
>
> $$\begin{aligned} File \;=\; \mu a.\, lookup;(found;a \\ &\Box\, \mu b.(query_q;b \\ &\qquad \Box\, reply_q \rhd fixOrder;a \\ &\qquad \Box\, \overline{abort};a \end{aligned}$$
>
> In this example, a file is either found after a local lookup or searched recursively. After a recursive call has been terminated a new search order for later searches is fixed. Finally, queries may be aborted any time. Here, the VPA property ensures that orders are modified only in the correct contexts and not, e.g., by a reply belonging to a different incarnation then the corresponding query.

Semantics. The difference between regular aspects and VPA-based aspects, i.e., stack-manipulating pointcuts, can be defined by two modifications to the semantics of regular aspects as outlined above. First, we have to add a call stack which is manipulated and tested as part of pointcut matching. Second, the notion of follow aspects has to be modified accordingly. Both of these issues can be conveniently addressed by translating a VPA-based aspect in a corresponding pointcut VPA that explicitly represents all traces in a consisting only of pointcut terms. Using the pointcut VPA the relevant functions of the operational semantics of the weaver shown in Fig. 3.1 can be redefined concisely: selection of applicable basic aspects (function **sel**) tests the VPA stack in case of constructors of the form f_s, $\underline{f_s}$ and determination of the follow aspect (function **next**) reduces to traversing one or (in the case of non-determinism) several transitions of the pointcut VPA. Finally, the advice to be applied once a pointcut (p, say) has been matched can be found simply by storing references to the basic aspect that contains the advice in the node of the pointcut VPA representing p. Apart from these changes, the weaving semantics for regular aspects does not have to be modified for VPA-based aspects. For details of the semantics see [NS06].

To conclude the discussion of VPA-based aspects, let us recall the similarity of properties of regular languages and VPLs, which, in particular, allows several important properties of aspects to be tackled equally well for regular and VPA-based aspects as shown in the following section (efficiency concerns notwithstanding).

[1] Matching of nested terms involving stack manipulating functions is a subject of future work.

3.2 Properties of aspects

While the precise definition of aspect mechanisms using formal semantics is a worthwhile goal of its own, methods to address the need for reasoning about properties of aspect-oriented programs are at least of equal importance. Aspects raise a number of specific problems concerning the enforcement, analysis or verification of AO properties. Two particularly important aspect-specific problems are the handling of interactions among aspects and the definition of conditions that delimit the effects of aspects on sets of base applications (as opposed to properties of aspects woven into a specific application). Aspect interactions constitutes one of the fundamental challenges for the composition of aspects and the latter can be seen as a generalization of traditional notions of encapsulation that are not appropriate anymore in the presence of aspects.

Relatively few approaches have been put forwarded for reasoning about aspectual properties in general and the two classes of properties singled out above in particular. We have been among the first to propose solutions for each of these two problems in form of, respectively, a notion of aspect interactions based on the simultaneous application of several aspects at an execution event and enabling the limitation of effects of aspects by ensuring that interactions are absent. Both of these proposals have been developed in the context of regular aspects and have been recently generalized to VPA-based aspects.

In the remainder of this section, we motivate our work on these two fundamental problems and explain its main technical characteristics. However, since aspect interactions and limits to how aspects may affect applications are intimately related (because the latter can be defined in terms of the former as discussed below), we first give an overview of the most relevant work.

Related work. Notable related work on aspect interactions include Dantas and Walker's [DW06] "harmless" aspects that guarantee not to cause interactions by means of a specially-tailored type system. Their approach provides only very limited support for direct reasoning about interactions. Another group of work has been presented on interactions due to modifications the static structure of OO programs, in particular Havinga et al [HNBA06] as well as Störzer and Krienke [SK03]. We are interested here in the more general and difficult problem of interactions resulting from runtime modifications by aspects. Furthermore, relevant work has also been done in closely related domains, in particular, feature interaction [FN03]. In typical settings of feature interaction, the interaction problem is alleviated in comparison to the realm of AOP because of the more homogeneous structure of telecommunication applications and the absence of certain aspect-specific structuring mechanisms, such as nested advice that may or may not call functionality of the base application of other aspects.

Very few work has aimed at mechanisms for the delimitation of effects of aspects. Aldrich [Ald05] has proposed a notion of modules (defined on the based of modules in the traditional, non AO, sense). The interfaces of these modules may export points which may be advised by aspects but can not be modified by external aspects otherwise. His approach does not support properties of aspects directly but rather allows to delimit their effects with respect to to individual modules. Kiczales and Mezini [KM05] have investigated the extension of method declarations by signatures similar to AspectJ pointcut declarations in order to make explicit the effect of aspects and thus enable informal and manual reasoning about modularity properties of aspects. In contrast, Goldmann and Katz [GK06] have presented work on the modular analysis of effects of aspects using model checking techniques. Finally, Skotiniotis et al. [SPL06] have recently proposed an extension to interfaces for adaptive programming that allows static conditions to be formulated on sequences of method calls that may crosscut the base application.

Finally, there are a number of articles presenting various categorizations for aspects with respect

to their effects on the base program and other aspects, notably by Katz [Kat06], Rinard et al. [RSB04] as well as by Clifton and Leavens [CL02a], all of which provide means useful for reasoning about the interactions and effect delimitation for certain classes of aspects. However, all of these approaches allow properties to be investigated only at a very coarse level, i.e., typically considering only properties of entire aspects.

3.2.1 Static analysis of interactions among aspects

Intuitively, interactions among aspects occur when the execution of one aspect modifies the behavior of another one. In the general case, such interactions may be caused by arbitrary modifications of aspects to any state that is accessible by different aspects, including state of the base application, modifications to files, and network communication. The definition and analysis of such general interaction properties requires, however, severe abstraction of the aspect and base programs, and is typically subject to limitations on the relationships between aspects that can be addressed (the approach of Goldmann and Katz [GK06], for instance, does not allow to apply aspects to other aspects).

Aspects are commonly used not as a mechanism for general programming but rather for the structured modification of base applications.[2] It seems promising to consider restricted notions of interactions that are specialized with respect to the aspect structure from such more specific uses. In particular, aspects commonly modify base applications at specific execution events where pointcuts match; some interactions can therefore be characterized solely by the simultaneous application of aspects at an execution event. It has turned out that this notion of interaction is of particular relevance because applications using several aspects frequently apply them in a pipelined fashion at common joinpoints, for instance, if three different aspects are used to compress, encrypt and log messages in a communication system (a programming technique at the heart of the so-called composition filters approach [BA01] to AOP).

We have provided the first formal definition of (potential) interactions in terms of the simultaneous application of aspects. In the context of regular aspects (cf. Def. 3.2 in Sec. 3.1.3) interactions occur when several basic aspects are weaved at the same joinpoint, i.e., when the aspect selection function **sel** returns more than one basic aspect $P \triangleright I$ in rule WOVEX of Def. 3.1 and hence rule ASPAPP is executed several times at one joinpoint.

> **Example:** In order to illustrate this notion of interaction, let us come back the example of queries in P2P applications introduced in Example 3.1.1 (see page 34). In addition to the aspect updating file information shown there, consider two other aspects: (i) an aspect that uses recursive file queries to eliminate certain stored files (e.g., illegal copies, (ii) an error aspect that raises exceptions when a reply occurs without any query having been initiated. In this case the former causes an interaction of file info updates, while the latter does not.

We determine interactions between two aspects using a static analysis of the independent ("parallel") execution of the aspects [5-DFS02b]. The analysis calculates a form of product of the two finite-state based expressions defining the two aspects. Concretely, it constructs a sequential result automata by merging paths of the input automata that consist of equal sequences of pointcuts. During this process basic aspects are marked that result in advice from both aspects to be weaved at the same event of the result automaton.

[2] The reader should keep in mind, though, that many aspect languages enable aspects to implement arbitrary programs: in AspectJ, e.g., the functionality of any Java program can be put in an aspect that advises the entry point of an otherwise empty base program — admittedly a technique rather akin to abuse than use of aspects.

[UN/FOLD] $\mu a.A = A[\mu a.A/a]$

[PRIORITY] $(P_1 \rhd I_1; A_1) \,\square\, (P_2 \rhd I_2; A_2) = (P_1 \rhd I_1; A_1) \,\square\, (P_2 \wedge \neg P_1 \rhd I_2; A_2)$

[COMMUT] $(P_1 \rhd I_1; A_1) \,\square\, (P_2 \rhd I_2; A_2) = (P_2 \rhd I_2; A_2) \,\square\, (P_1 \rhd I_1; A_1)$
; if $P_1 \wedge P_2$ has no solution

[ELIM] $P \rhd I = \mathit{false} \rhd I$ *;if P has no solution*

[PROPAG]
$$\begin{array}{llcl} \text{let} & A & = & (P_1 \rhd I_1; A_1) \,\square\, \ldots \,\square\, (P_n \rhd I_n; A_n) \\ \text{and} & A' & = & (P'_1 \rhd I'_1; A'_1) \,\square\, \ldots \,\square\, (P'_m \rhd I'_m; A'_m) \\ \text{then} & A \parallel A' & = & \square_{i=1..n}^{j=1..m} P_i \wedge P'_i \rhd (I_i \bowtie I'_j);\, (A_i \parallel A'_j) \\ & & & \square_{i=1..n} P_i \rhd I_i;\, (A_i \parallel A') \\ & & & \square_{j=1..m} P'_j \rhd I'_j;\, (A \parallel A'_j) \end{array}$$

Figure 3.2: Laws for aspects

Technically, this analysis is defined by an equational system consisting of 8 equations that enables two independent aspects, noted $A_1 \parallel A_2$, to be transformed into an equivalent sequential aspect with marked interactions. Fig. 3.2 shows an excerpt of this system. Marking of interactions is achieved by rule PROPAG that is used to 'push' the parallel operator inwards into expressions, thus making them more deterministic. The topmost row of the result expression $A \parallel A'$ corresponds to the case of interest, that is, all the combinations of initial pointcuts P_i from A and P'_j from A' both of which have to match: in this case, a (potential) interaction is recorded in the result using the expression $I_i \bowtie I'_j$ over the corresponding advice. The two other rows in the result expression group all subexpressions in which only a pointcut of one of the argument aspects matches and therefore cannot give rise to interactions. The remaining rules allow to un/fold recursive definitions (rule UN/FOLD). Two rules govern choices: rule PRIORITY expresses that the choice is deterministic because the first branch has priority over the second (note that this rule has to be slightly changed in order to take variables into account, see [5-DFS04a] for details), COMMUTE states commutativity of choices if none of the two pointcuts at the head of the branches of the choice match. Rule ELIM, finally, shows an elimination rule that is helpful to eliminate branches which cannot be taken.

The notion of interactions through multiple aspects applicable at common execution events can naturally be extended to VPA-based aspects and applied to the balanced contexts that VPAs support. Since the intersection operation on VPAs is closed and the test for emptiness is decidable, the interaction analysis between VPA-based aspects A_1, A_2 can be reformulated for VPAs in terms of the standard operations for VPAs V as $A_1 \cap A_2 \neq \emptyset$ (and a corresponding marking of interaction points) [NS06].

We have developed a prototype implementation of this analysis. While this implementation also allows to analyze interactions between regular aspects, we are currently working on a more efficient implementation for the regular case using model checking.

To conclude the discussion of aspect interactions, let us note that these techniques yield interactions between two aspects, that is, independent from their application to any base program (this has been termed strong independence in [5-DFS02b]). It is, however, also be possible to abstract the base program into a regular or VPA-based abstraction and then considering interactions only that may occur during executions of the abstracted base program (weak independence in [5-DFS02b]).

3.2.2 Applicability conditions for aspects

We now turn to the problem of techniques for determining and limiting the effects of aspects. Reconsidering the discussion of related work above, the following characteristics seem desirable:

- Support for the expression of crosscutting effects to arbitrary entities of base programs, including to an existing module structure (but not restricted to existing module structures as the approaches by Aldrich [Ald05] or Skotiniotis et al. [SPL06]).
- Formal and if possible automatic support for the enforcement of limiting properties (thus improving on Kiczales and Mezini's informal and manual approach [KM05]).

A general notion of effects of aspects meeting these characteristics can be defined in terms of applicability conditions[3] for aspects and base programs. Concretely, applicability conditions can be defined as regular aspects that raise exception events in situations that are deemed to be erroneous. An interaction between an applicability-defining aspect and a second aspect then indicates that the second aspect is not compatible when it would trigger advice in erroneous situations. Furthermore, based on a regular abstraction of the base program, interactions between a class of base programs (B, say) and an applicability condition (a, say) can be determined: if there are no interactions, any aspect that is compatible with a can be woven with any base program of B and is guaranteed not to engage in any of the erroneous situation represented by a. Finally, in the case that an aspect is not compatible with an applicability condition it is possible to weave it nevertheless but also weave the applicability-defining aspect at the same time, such that exceptions are raised during execution in erroneous situations.

> **Example:** In the P2P setting, we could use applicability conditions, for example, to ensure that any query can only occur after the network has been reasonably set up, e.g., by appropriate initialization of the corresponding infrastructure.

We have shown that interaction analyses as discussed in the previous section can be applied in this context. Taking into account applicability conditions for interaction analysis yields the notion of "contextual" independence [5-DFS04a], which is finer than strong independence (because it takes into account some traces of the base execution) but coarser than weak independence (because it does not take into account all base program traces).

To conclude the discussion of applicability conditions for aspects, let us note that they meet the two characteristics introduced in the beginning: they can define properties of fine-grained entities and come equipped with automatic support for analysis and enforcement.

3.3 Aspect composition

The construction of any program (using aspect-oriented techniques or not) can be seen as the application of some composition operation to some basic entities. Functional programming, for instance, is based on the combination of functions using functional composition. Component-based software construction relies on the provision of glue code between software components. Aspect-oriented software development fits this picture well: aspects can be considered as the basic entities that have to be composed with the base program and one another.

This perspective on AOSD reveals three requirements of aspect composition beyond the basic observation that aspects have to be weaved, typically invasively, into a base application:

[3] Called "aspect requirements" in [5-DFS04a].

1. Aspect composition should enable interactions, the main problem for aspect composition, to be resolved.

2. Aspects should be composable in a flexible manner, in particular, composition should not be restricted to aspects as a whole.

3. Composition should be supported through the use of explicit well-defined composition operators that allow to express much more general relations than common precedence specifications. Furthermore, they should support reasoning over compositions.

Currently, most aspect systems support aspect composition only in limited ways and frequently in a very coarse-grained fashion (typically by composing aspects without modification as a whole) or through very fine-grained mechanisms. In AspectJ, for example, aspect precedence specifications can be used to compose aspects as a whole and the `adviceexecution` pointcut predicate may be used to define if and how an aspect should be applied in the presence of other aspect at individual joinpoints. There are only few structured mechanisms that allow parts of aspects to be composed or that allow composition of aspects to be limited to parts of the execution of the woven program. Work in this direction has been pursued mostly based on specific aspect models, e.g., by Bergmans et al. [BA01] who define aspect compositions in terms compositions of stateless filters that are applied when messages are sent. Lieberherr et al. [LOML01, LLO03] have introduced aspectual collaborations that allowed aspect composition to be defined in terms of specialized merging operations on class hierarchies. The Hyper/J approach [TOHS99], similarly, allowed general, unstructured composition programs to be defined for the composition of aspects (called hyperslices). These approaches fall short with respect to the requirements above because they do not make the resolution of aspect interactions explicit, are not operator-based, and, for some of them, are not flexible enough.

Regular aspects admit a flexible notion of *composition adaptors* [5-DFS04a, 5-DFS02b]. These operators are defined as a restricted form of regular aspects, see Fig. 3.3a, that specify unary or binary advice transformers that are applied according to a finite-state automaton. These operators meet the three requirements introduced above. Compositions are performed flexibly only when a program execution matches the finite-state based composition definition. Conflict resolution can be performed using binary transformers by, e.g., reordering advice, exempting advice from application or replacing advice altogether. Finally, the interaction analyses admitted by regular aspects can be used also in the presence of composition operators, essentially by replacing the propagation rule of Fig. 3.2 by the rule shown in Fig. 3.3b that applies the appropriate (unary or binary) composition transformer where necessary.

A feature of composition adaptors that is particular attractive from a software engineering viewpoint is that composition operators may therefore be used as part of an *iterative incremental development method*: base programs, application conditions and regular aspects may be analyzed for interactions using the different static analyses presented in the previous section, followed by the resolution of some of the interactions using composition adaptors. The resulting system can be analyzed once again, etc.

Finally, let us briefly mention a second notion of flexible, operator-based aspect compositions we have investigated. In the context of an instantiation of the EAOP model that featured turing-complete pointcuts and advice implemented as sequence of Java statements [6-DS03, 9-DS02], aspect composition can been defined explicitly based on a (directed acyclic) graph structure whose nodes are composition operators and leaves are aspects. Aspects are composed by constructing or modifying composition graphs. The composition structure allows for arbitrary flexible aspect composition because the composition can be modified dynamically. We have proposed four composition operators to

$$
\begin{array}{lcll}
O & ::= & \mu a.O \mid C \rhd F; O \mid C \rhd F; a \mid O_1 \,\Box\, O_2 & \\
 & & & \\
F & ::= & (U \oplus B) & \textit{; pair of transformers} \\
U & ::= & \mathit{id} \mid \mathit{skip} & \textit{; unary transformers} \\
B & ::= & \bowtie \mid \mathit{seq} \mid \mathit{fst} \mid \mathit{snd} \mid \mathit{skip} & \textit{; binary transformers}
\end{array}
$$

(a) Syntax

$$
\begin{array}{llcl}
\text{let} & A & = & (C_1 \rhd I_1; A_1) \,\Box\, \ldots \,\Box\, (C_m \rhd I_m; A_m) \\
\text{and} & A' & = & (C'_1 \rhd I'_1; A'_1) \,\Box\, \ldots \,\Box\, (C'_n \rhd I'_n; A'_n) \\
\text{and} & O & = & (C^O_1 \rhd (u_1 \oplus b_1); O_1) \,\Box\, \ldots \,\Box\, (C^O_o \rhd (u_o \oplus b_o); O_o)
\end{array}
$$

$$
\begin{array}{llcl}
\text{then} & A \parallel_O A' & = & \Box_{i=1..m,\, j=1..n, k=1..o} C_i \wedge C'_j \wedge C^O_k \rhd b_k(I_i, I'_j);\, (A_i \parallel_{O_k} A'_j) \\
 & & & \Box\ \Box_{i=1..n}^{j=1..m} C_i \wedge C'_j \rhd I_i \bowtie I'_j;\, (A_i \parallel_O A'_j) \\
 & & & \Box\ \Box_{i=1..n}^{k=1..o} C_i \wedge C^O_k \rhd u_k(I_i);\, (A_i \parallel_{O_k} A') \\
 & & & \Box\ \Box_{i=1..n} C_i \rhd I_i;\, (A_i \parallel_O A') \\
 & & & \Box\ \Box_{j=1..m}^{k=1..o} C'_j \wedge C^O_k \rhd u_k(I'_j);\, (A \parallel_{O_k} A'_j) \\
 & & & \Box\ \Box_{j=1..m} C'_j \rhd I'_j;\, (A \parallel_O A'_j)
\end{array}
$$

(b) Interaction propagation

Figure 3.3: Composition adaptors for regular aspects

address aspect interactions in this aspect model, either by ordering aspects that apply at a same joinpoint or by conditionally execute one aspect if another one has (or has not) previously been applied.

3.4 Conclusion and perspectives

In this chapter we have presented different approaches to provide aspects with formal semantics in order to define them rigorously and support formal and, for some of them, automatic reasoning about properties of AO programs. We have introduced semantics and reasoning methods for aspect languages of different expressiveness, ranging from regular aspects to turing-complete pointcut languages. Based on this foundational work on AOP, two of the most important problems of AOP, aspect interaction and applicability conditions on aspects, have been addressed. Finally, we have shown that these approaches support flexible and declarative notions of aspect composition that allow to resolve aspect interactions.

Perspectives. The different approaches presented in this chapter pave the way for further studies on various questions concerning the foundations of AOP.

Most importantly, the notions of applicability conditions and aspect composition operators that we have introduced should be useful as a basis for a comprehensive study of *modular properties of aspects*, in particular, the relationship of traditional modules (that feature strong encapsulation properties, support separate compilation, etc.) and crosscutting functionalities. Since this is probably the single most important issue pertaining to the foundations of AOP, we discuss it as part of the major perspectives of our work in detail in Sec. 6.

As we have discussed our notion of aspect interactions is limited in the sense that it does not capture interactions arising from the manipulation and use of some shared state at different execution points. While our analysis is not amenable as is to handle corresponding *more general notions of aspect interactions*, regular aspects as well as VPA-based aspects should be well-suited to a study of more general interactions. Regular aspects, in particular, should be amenable to support corresponding analyses based on model checking.

There is a large range of other classes of different expressiveness than those we have considered. There is for instance a large class of counter automata that allow to represent many context-free programs and some context-sensitive ones, while supporting automatic analysis of properties that are relevant for aspects, their compositions and interactions. Furthermore, VPAs are currently investigated by many research groups, in particular, develop more efficient analyses of their properties. Results on these issues could be usefully integrated into aspect languages by extending the framework for formal definition and properties presented here.

Chapter 4

Expressive aspects for component-based and systems programming

While rigorous semantics of aspect languages and systems definitely is a fundamental and intellectually challenging problem of AOP, the development of aspect languages has always been driven by the quest for concrete means for the modularization of crosscutting concerns. Right from the inception of AOP as a research domain of its own, in particular through the work initiated by G. Kiczales at Xerox PARC [KLo97], this approach has allowed to generate wide-spread interest by scientists and practitioners alike because of its focus on deficiencies of widely used programming methodologies and resulting problems in the structuring of, mostly sequential, large-scale applications.

Two domains in the field of software engineering that are particularly challenging — and potentially rewarding — for the application of AO methods are component-based and systems programming. Apart from the fact that both are subject to a large number of crosscutting concerns, they raise intricate issues related to the integration of AO technology with well-established development and programming methodologies. Component-based software development is built on the premise of the strong encapsulation properties of software components [SGM02], which seems to contradict the use of aspects to modularize concerns impacting large sets of possibly otherwise unrelated components. Expressive aspect languages are potentially useful in this context because they can potentially denote scattered execution events while at the same time explicitly represent constraints on these events stemming from the encapsulation properties of components. Systems programming on the other hand is characterized by execution efficiency being a vital condition. This impetus has important repercussions on the use of AO methods in this domain: for one, many techniques to support aspect languages, such as delegation-based techniques, cannot be reasonably used; for another, systems programming often disregard regular solutions in favor of more efficient but ad hoc ones. While the latter in principle argues for the use of expressive aspect languages (because they should be better suited to refer to such ad hoc structures), the former casts serious doubts on the applicability of expressive, and possibly costly, aspect mechanisms in the domain of systems programming.

In this chapter we present results pertaining to these modularization problems in these two domains. Regarding software components we have investigated component interfaces that include explicit protocols to address modularization problems in the presence of strong encapsulation in a precise manner. Our work is distinguished from previous approaches by the integration of more expressive protocols than commonly used with components. As to aspects for system-level programming, we have focused on an instantiation of the EAOP model to an aspect language for base programs implemented using the C programming language. The resulting aspect language which has been integrated

in the Arachne system is still unique in providing non-atomic aspects for C and supporting a dynamic weaving strategy, that is, weaving of aspects into native code during execution. Finally, the composition and aspect languages developed for both domains have been formally defined, yielding, in particular, the first formal semantics for an aspect systems for C applications.

The results presented in this chapter have mostly been achieved as part of different collaborations. Part of the work on protocols, aspects and software components has been performed in collaboration with my former PhD student Andrés Farías [Far03] and the PhD thesis by Ha Nguyen that I am currently supervising. The instantiation of the EAOP model to the Arachne aspect language has been done in cooperation with Marc Ségura-Devillechaise during his, now finished, PhD thesis and my colleagues Jean-Marc Menaud and Rémi Douence from EMN.

Albeit we do not elaborate on this work in this thesis, let us briefly hint at an important relationship between component-based and system-level programming that is essentially supported by expressive aspect languages. Expressive aspects are, in fact, a useful tool to refactor systems applications to make explicit the interfaces governing key functionalities of these applications. We have substantiated this claim by developing an aspect language for the construction of component-based interfaces of system-level code [5-MLMS04]. In this work, an aspect language allows the identification of functions that have to be part of interfaces using temporal logic based pointcuts that quantify over execution paths and the interfaces are defined using advice.

The remainder of this chapter presents the results in these two domains. Section 4.1 discusses the use of expressive protocols in software components and their manipulation using aspects. Section 4.2 discusses the main characteristics of the design and implementation of the Arachne aspect language for C.

4.1 Components with explicit protocols

Component-based software development (CBSE) today constitutes the main approach to the construction of large-scale applications. Software components support the incremental development of large applications through strong encapsulation of software artifacts that enables the assembly of applications from third-party components and their execution in different deployment contexts. By leveraging this development method to build large software systems from sets of pre-fabricated components, CBSE promises to lift the currently still predominant artisan software development methods to an engineering field enabling the construction of provably correct software systems from modular units [McI68].

Technically, the main feature underlying the notion of software components is that they rely on "contractually specified interfaces and explicit context dependencies only" [Szy98]. The absence of implicit context-dependencies — which arise, for instance, if a component directly accesses some state variable of another one without using a method of the second component's interface for that purpose — enables components to be considered self-contained entities whose composition properties can be completely defined in terms of its interface.

Currently, most approaches to software components, in particular all so-called industrial component platforms, such as EJB, .NET et Corba Components, employ a notion of interface that only defines the signatures of services, i.e., their names, types of arguments and results. Such interfaces do not make explicit the semantics of the services exposed by the interfaces: this is an important impediment to the goal of independent provably-correct assembly of software components, because applicability conditions for software services are easily violated, e.g., the condition that an initialization has to be performed before a specific service can be called. Furthermore, basic component

correctness properties, such as compatibility (which states that two composed components correctly use their respective services) and substitutability (which states that a component may be used in all contexts where another one can be used), cannot be verified, let alone more complex properties, such as absence of deadlock. These limitations also hold for many academic component models, for instance the Fractal component model [BCL+06, Fra][1] to name just one.

A well-known remedy to this lack of explicit semantic information about components are protocols that make explicit constraints on admissible sequences of services. Frequently, protocols are defined in form of additional specifications at the design level, for instance, using statecharts, sequence diagrams or OCL specifications that are part of UML. There are, however, also several approaches that integrate protocols into component interfaces on the language level, see, e.g., [PV02]. Almost all of these approaches use finite-state based protocols, which enables component-relevant properties, such as compatibility, substitutability and deadlock, to be defined precisely and to be supported by automatic tools, in particular through model checking. The relatively few approaches featuring non-regular protocol languages, for instance, protocols based on symbolic transition systems [HL95], suffer from severely restricted support for reasoning about protocol properties that is essential in a component-based setting.

Our work in this domain has focused on more expressive notions of component protocols featuring either finite-state based, i.e., regular, protocols with richer structure, non-regular protocols and the construction and manipulation of component protocols using aspects and composition operators. Concretely, we have defined a component model, Components with Explicit Protocols (CwEP) that make explicit sets of component ids and provide concise support for certain communication patterns. Furthermore, modularization of crosscutting concerns is supported through aspect languages that manipulate such protocols by means of protocol composition operators. A distinguishing feature of this approach to component construction is that component properties can be inferred from interface protocols and the operators used to manipulate them; in some cases, properties of components can even be inferred solely from the properties of the operators, i.e., independent from the argument protocols. Furthermore, we have defined an aspect language for the manipulation of such protocols and developed a framework-based implementation of this component model that integrates smoothly with the Enterprise JavaBeans platform.

4.1.1 The CwEP model

The CwEP model has been defined in order to investigate precise means for component composition by providing components with a rich notion of interfaces consisting of three parts [Far03]: (i) a set of services signatures, (ii) a protocol governing interactions of that component and (iii) a set of component identities that can be used to restrict the availability of services.

The protocols of CwEP components are defined as finite-state automata whose transition labels denote directed service names, that is, distinguish service requests from and service calls to other components, similar to Yellin and Strom's seminal work [YS97]. Acceptance of individual transitions may further be restricted by constraints on component identities of collaborating components. Such constraints are expressed in terms of sets of identities that may request services or to which service calls are issued. Furthermore, multicast communication to or from all components belonging to an identity set can be denoted using a particular kind of transition. As we have shown, the resulting protocols are equivalent to finite-state protocols but allow significantly more concise definitions of certain classes of component protocols, specifically in the case of publish-subscribe style [5-FS02].

[1] The Fractal model includes, however, powerful mechanisms to reflect on component behaviors that allow to enrich component interfaces by, e.g., adding protocol-enforcing behavior.

Composition properties. Fundamental properties of component composition can be defined in terms of interfaces. Two properties that are of particular importance in this context: compatibility and substitutability. Compatibility is the relation between components that ensures that two components can be used together in isolation without inducing errors, e.g., interactions between the two collaborators are deadlock free. Substitutability is the basic relation ensuring that a protocol may be replaced in all contexts by another one without giving rise to errors; formulated differently, replacing the protocol still allows all services requested by collaborators to be served successfully.

Components with explicit protocols allow these properties to be defined in a fine-grained manner. A large number of different relations between execution traces have been proposed that can serve as a suitable basis, depending on whether only traces are only taken into account or if failures are also considered, the expressiveness of the language used for protocol definitions, etc.

We have explored a definition of these two properties based on work by Nierstrasz [Nie93] who defines notions of compatibility and substitutability in terms of traces and failures for objects having interfaces consisting of (unrelated) sets of method signatures. We have extended his definitions to CwEP components, in particular, accommodating directed service requests, component identities and broadcast operations to sets of collaborating components. More recently, we have considered compatibility and substitutability of components with non-regular protocol language [5-Süd05]. In this context, protocols can use counters whose manipulation is restricted using an effect type system that enables many context-free and even some context-sensitive protocols to be expressed [Pun99]. We have shown that our protocol language for components admits compatibility and substitutability properties analogous to the regular case.

Protocol construction operators. Most approaches to the definition of software components focus on programmatic means for the construction of components and compositions from entities, such as individual methods, that do not possess composition properties by itself. With such approaches, properties of compositional systems are then ensured using analysis, verification or validation of composites, that is, after system construction time. In contrast, we have aimed at a constructive approach to composition that allows to ensure composition properties by construction of components with explicit protocols.

We have formally defined a set of component constructor and composition operations and investigated to what extent these operators preserve compatibility and substitutability properties of components. In the CwEP model such operators may modify the three parts of interfaces mentioned above: (i) they may modify the set of services offered by components (e.g., to add or disable services), (ii) may modify the protocols governing components (e.g., by adding branches) or (iii) modify the set of component identities governing the execution of transitions.

We have defined a set of 12 operators that are useful to manipulate these parts. They include, among others, a union operation on protocols that can be used to add new branches to protocols, an insertion operation allowing to insert a protocol at a given state into another one, and operators that propagate given component identities over parts of protocols. We have shown that straightforward definitions of such operators frequently yield results which preserve component properties as far as the set of accepted traces is concerned but yield sets of failures that are larger than necessary because of alternative paths with common prefixes that yield to transitions with different failure sets whose union is the failure set required for compatibility and substitutability properties to hold. We have defined a technique to reduce the set of failures to the minimum necessary by defining an equivalence relation identifying such prefixes of equal traces [5-FS02, Far03].

Example: The resulting composition operators may observe composition properties generally

or in some contexts only. The union operation on protocols, for instance, preserves all traces and all failures (except for the start state) of the argument protocols, and therefore also preserves component compatibility. As a second example, concatenation of two protocols always preserves compatibility with a protocol compatible with the first argument of the composition but never preserves substitutability because the failures in final states of the first argument protocol and the corresponding states of the concatenation differ.

4.1.2 Manipulation of protocols using aspects

Besides their expressiveness, component protocols are useful as a base structure to which aspects can be applied. Through the manipulation of protocols, aspects can modularize crosscutting concerns of components through modifications to the relationships governing service requests into and out of a component, that is, without violating the black-box encapsulation properties of components. This approach improves on the currently predominant approach to crosscutting in component-based systems that apply atomic aspects to individual services, see, e.g., [DEM02, POM03, CC04].

The expressiveness of a pointcut language over protocols is, in principle, independent from the expressiveness of the protocol language. However, it is often useful to use an aspect language to be of equal expressiveness to the protocol language, because pointcuts can then exactly denote arbitrary traces generated by the execution of the protocol. Advice may be used to apply operators that introspect and modify protocols. In particular, dynamic modifications to the structure of a protocol (i.e., the base structure of aspects over protocols) are frequently useful, for instance, to model the creation or closing of services of components. This last property is very uncommon in mainstream aspect-oriented programming: in AspectJ, e.g., modifications to the program structure via so-called intertype-declarations cannot be performed at runtime and many aspect systems do not know base structure modifying mechanisms at all.

We have explored aspect languages for the modification of component protocols for the cases of finite-state and VPA-based protocols. For finite-state protocols, we have defined an aspect language [Far03] whose pointcut language extends the regular aspects of Def. 3.1 (see page 31) by several constructors for regular pointcuts. This aspect language includes different protocol modifying operators from those introduced at the end of Sec. 4.1.1. The semantics of this language can therefore be defined akin to that of regular aspects as described in Sec. 3.1.3.

However, runtime manipulations of the structure of protocols requires special attention. Unrestricted use of this feature easily causes semantic problems. For example, the application of advice that adds a new path to a protocol can lead to non-terminating weaving if the same advice is applicable on the newly introduced path. Apart from analyzing complete aspect and base programs for such situations, we have devised two different general solutions to this problem: (i) employ protocol operators for which the absence of termination problems can be proved once and for all independent from the argument protocols, (ii) restrict the weaver semantics such that behavior introduced by advice is not subject to weaving. We have proven the correctness of the latter solution by a proof [9-FS04] based on an extension of the weaver definition shown in Fig. 3.1 (see page 35).

Recently, we have started to explore aspects over VPA-based protocols [NS06], see Sec. 3.1.4. We have shown that dynamic modifications to the structure of protocols pose similar problems as in the regular case and can be solved using the same techniques.

4.1.3 Industrial component models and AOP

The so-called industrial component models, such as Enterprise JavaBeans (EJBs) [Inc03] and CORBA Components [Gro02], for distributed applications, or more precisely 3-tier applications deserve particular attention in a discussion on aspects and components. First, crosscutting concerns are of highest importance in these models: the EJB specification [Inc03], for instance, defines in detail how to handle three crosscutting concerns — persistence, transactions and security — on around 75% of its 646 pages. Second, 3-tier applications constitute the most important application domain for AOP currently: Spring AOP [Spr], the AspectJ-like framework for atomic aspects in the Spring framework for the development of EJB-based applications, has a higher usage count nowadays than AspectJ.

However, the frameworks for the implementation of crosscutting concerns in these component models fall far short from providing satisfactory modularization mechanisms for the concerns. As discussed in our detailed analysis [1-NDS05], the industrial component models are subject to three major deficiencies with respect to this modularization issue.

- Incomplete modularization
- Limited concern models
- Lack of extensibility of concern models

Enterprise JavaBeans, for instance, support the implementation of security concerns through a role-based access control model which supports modular definition using so-called deployment descriptors. However, modularization is only partial because EJBs also depend on a low-level interface for access control whose use results in the scattering of access control instructions. Furthermore, EJBs security model is subject to severe restrictions even in comparison to established role-based security models [RSC+96], let alone more recent state-of-the-art approaches, such as information flow based approaches to security [ML97]. Finally, bean developers do not have any possibility to extend the underlying security model.

A fair number of aspect-oriented approaches as well as approaches using based on metaprotocols have been put forward to remove in particular the first restriction. As shown by our comprehensive comparison [1-NDS05], such approaches allow to significantly improve the modularization of crosscutting concerns while preserving compatibility with the main characteristics of the underlying component model, in particular, their client-server architecture, black-box access to components and container-based execution models.

The CwEP model allows to further improve the modularization of crosscutting concerns in industrial component models. Since CwEP protocols allow to express relationships between different components in protocols, concerns can be described more concisely than with previous atomic aspect systems but also the few with non-atomic features, such as AspectJ2EE `remotecall` pointcut [CG04].

We have proved the compatibility of the CwEP model with industrial component models through the development of an implementation of the CwEP model that integrates smoothly with the Enterprise JavaBeans platform [Far03]. Concretely, this implementation augments the EJB container abstraction by a manager component that allows runtime events to be forwarded to entity or session beans with explicit protocols. The latter can be implemented using a framework providing support for component identities, implementations of the constructor and composition operators as well as specialized implementations of the substitutability and composability algorithms. We have shown, in particular, how CwEP beans facilitate the implementation of security policies in EJBs [5-FS02].

4.2 Expressive aspects for system-level applications

A second major focus of my work on expressive aspect languages for sequential programs, has been aspect languages for applications written in C and C++. This work has been motivated by the observation that many crosscutting concerns of system-level applications, for instance manipulations of communication protocols in networking applications or low-level access to data structures, benefit from a non-atomic aspect model [5-DFL$^+$05, 3-SDML$^+$06].

While the importance of non-atomic relationships on the aspect language level for system-level programming has been discussed in 2001 [CKFS01], no aspect systems have provided language support for such relationships and corresponding weaver technology prior to our work. The other approaches for aspects in C-like languages, most prominently Aspect-C++ [SGSP02], feature an atomic pointcut language essentially transposing the AspectJ aspect model. They are therefore subject to the problems introduced in Chap. 2.1 regarding the concise formulation of non-atomic crosscutting relationships. Furthermore, none of these system has been given a formal semantics in order to precisely investigate aspect properties.

We have realized a non-atomic aspect model by extending the Arachne system for AOP in C into an instantiation of the EAOP model. The original Arachne system (initially developed under the name μDyner [SDMML03]) offered an atomic aspect model (as usual, cflow relationships notwithstanding). It marks an interesting design point within the space of aspect languages and implementation mechanisms that sets it apart, for instance, from typical Java-based aspect systems because it features dynamic weaving of aspects into executing, i.e., native, C code. Dynamic weaving into C applications is often useful for system-level applications because it allows applications to be manipulated without access to their source code and without stopping them, i.e., without incurring downtime, which is particular important in widely-used server applications. However, weaving into native code imposes strong restrictions on the language and the implementation level. On the language level, certain base program entities that are easily accessibly as part of aspect systems that weave into source code, for instance Aspect-C++, cannot be accessed during execution of native code: this is the case, for instance, for variables that are local to nested blocks in the source code. As far as execution support is concerned, only very small runtime overhead, in particular due to weaving, is typically tolerated for system-level applications. The original Arachne implementation weaved new calls to C-functions with an overhead of a factor of around 2 compared to plain C function calls, relative overhead which was 10–25 times smaller than that of typical dynamic aspect systems for Java.

We have therefore designed and implemented an instantiation of the EAOP model that adds finite-state based pointcuts for C programs to Arachne's initial aspect model while preserving its performance characteristics. In the remainder of this section, the major language and implementation issues are presented as well as our approach to giving formal semantics for the extended Arachne system is presented.

While we do not detail this work here, let us note that the resulting system has been applied to the problem of modularization and dynamic evolution of medical image generation software for, e.g., tomographs, in the context of a cooperation with Siemens AG, Munich, Germany [5-FSS$^+$05].

4.2.1 Expressive aspects for C: language and implementation

Originally, the Arachne approach featured an aspect language providing a set of primitive pointcut constructors to match function calls, accesses to global variables, and to test for control-flow relationships. Advice allowed to call arbitrary C functions defined as part of aspects or in the underlying base applications.

seq(**call**(*void* $*$ *malloc*(*size_t*)) && **args**(*allocatedSize*) && **return**(*buffer*) ;
write(*buffer*) && **size**(*writtenSize*) && **if**(*writtenSize* $>$ *allocatedSize*)
then *reportOverflow*(); $*$
call(*void free*(*void*$*$)))

Figure 4.1: Buffer overflow detection aspect in Arachne

We have extended this aspect language by two features: finite-state based aspects and aliases to global variables. The former allows to match sequences of primitive pointcuts and enable information to be passed from the different constituents of a sequence to an advice. The latter provides access to a class of local entities in C programs that are frequently used in system applications and that, in contrast to local variables, can be accessed and manipulated during execution of native code using standard runtime systems.

A direct approach to implementing non-atomic aspects by interpreting finite-state based pointcut definitions as introduced earlier in this thesis by regular aspects and implemented, for instance, in the CwEP approach, is not appropriate in the context of dynamic weaving of C applications for different reasons. First, in order to be able to manage variable references appropriately, sequences must have unambiguous start steps. Nested regular pointcuts and aspects may also give rise to ambiguities that impeded the correct management of variable references.

For these reasons, we have defined a limited notion of regular aspects that permits to accommodate the variable references manageable on the level of native code. These limited regular aspects are called "sequence" aspects in Arachne, noted the constructor **seq**, because of how they are typically used. Sequences start with a disjunction of primitive aspects, i.e., function call or variable access pointcuts, may be followed by a number of disjunctions of primitive pointcuts that may be arbitrarily often repeated and terminate with a single disjunction of primitive aspects. A pointcut at any step in the sequence may trigger advice consisting of a single function call.

> **Example:** Figure 4.1 shows an example aspect for buffer overflow detection defined in the resulting aspect language. It defines a sequence of three steps that match calls for allocation, repeated write operations on buffers, and a call to deallocate buffers, respectively. At allocation time, the size of the allocated memory is recorded and compared with the size of data written at the second step. If more data is written than has been allocated an overflow is reported by applying the advice that follows the keyword **then**. Here, the pointcut **write**(*buffer*) matches accesses to global variables as well as aliases to those variables.

The sequence aspect construct has been implemented using a list that links instructions on the native code level that correspond to successive steps in the sequence. Once a step is matched and the corresponding advice executed (if there is one), a residue computation updates the list to reflect eventual changes in the instructions belonging to the sequence, e.g., introducing back edges if a sequence step can be repeated. Using this implementation approach, the cost for sequence aspects is proportional to the number of steps in the sequence and is circa three times more costly than corresponding sequences of plain C calls [3-DFL$^+$06]. Accesses to data are comparatively very costly during execution of native code because they require complete memory pages to be locked and corresponding handler code to be executed. Our implementation performs these different steps with an overhead of approximately 9700 processor cycles (compared to 1 processor cycle for a plain C access), almost all of which is, however, due to page locking, a requirement any dynamic weaver for C is subjected to.

$$\begin{array}{l} \textbf{seq}(\ p_1\ \textbf{then}\ f_1; \\ \quad p_2\ \textbf{then}\ f_2;\ [\ *\ ] \\ \quad p_3\ \textbf{then}\ f_3;\ \ ) \end{array} \quad \equiv \quad \begin{array}{l} \mu a_1.\ C_{p_1} \rhd f_1; \\ \quad (a_1 \mid\mid \mu a_2.\ (\ C_{p_3} \rhd f_3;\ \text{STOP}\ ) \\ \qquad\qquad \Box\ C_{p_2} \rhd f_2; a_2) \end{array}$$

Figure 4.2: Definition of sequence pointcuts

4.2.2 Formal semantics

A second open question we have addressed as part of our work on the Arachne system is the question how to give formal semantics to such systems in order to foster better understanding of aspect systems in this domain and support reasoning about them. In addition to the challenge of defining a semantics at a high level of abstraction, a specific challenge of C-based programs consists in certain idiosyncrasies pertaining to the C-based compilation schemes. The use of macros as integral part of C programs, into which Arachne aspects are compiled, in particular, has to be accounted for.

We have addressed these challenges by defining to different formal semantics for the Arachne system:

- A high-level semantics [5-DFL$^+$05, 3-DFL$^+$06] defining the main features by a translation of Arachne's aspect language into an extension of the language for regular aspects shown in Fig. 3.2 (see page 34).
- An implementation-level semantics [3-DFL$^+$06, Fri05] defining the weaving of Arachne aspects by a source-to-source transformation of base applications into woven programs that are expressed as C programs.

High-level semantics. The high-level semantics abstracts from all implementation issues relating to the Arachne's weaving strategy of aspects into executing native code. In return, it defines the main properties of Arachne's aspect language in very concise and well-understandable form. Concretely, it extends the language of regular aspects by allowing the use of parallel combinations of aspects in the scope of the sequential operators (repetition, choice and sequence).

Figure 4.2 shows the definition of sequence aspects [5-DFL$^+$05] that makes explicit three important properties. First, it defines that a new instance of the sequence is created as soon as a match for the first primitive pointcut p_1 is found by looping on a_1 in parallel to the on-going match on p_2 that is performed by the original instance of the sequence aspect. Second, an aspect instance is terminated (by the primitive aspect STOP) after the last sequence step has been executed. Third, repetitive steps (i.e., steps in the scope of the '$*$' operator) yield control to the following step, if both the pointcuts of the repetitive step and the following step match (this follows from the fact that the first argument of the choice operator $\Box$ of regular aspects has priority over the second one).

This semantics allows equational reasoning to be applied in order to manually proof equivalences between aspect expressions [3-DFL$^+$06]. The third property of the sequence aspect mentioned above can, in fact, be proven using induction of the steps of a sequence.

Implementation-level semantics. The high-level semantics above abstracts from all implementation details, in particular, the C code generated by the Arachne compiler (that is then compiled by the gcc compiler), Arachne's runtime libraries and its weaving strategy (that is essentially based on rewriting of native code). A precise definition of these phases is necessary to establish the correctness

SA: SequenceAspect $\longmapsto$*INT* $\longmapsto$*CG*$*$

```
SA⟦seq(AspPrim AspSeqElts AspSeqElt)⟧(n) =
        let x = NUM⟦AspSeqElts⟧(2)
            v = concat(VARs⟦AspPrim⟧,
                       concat(VARs⟦AspSeqElts⟧,VARs⟦AspSeqElt⟧))
            in defineAliasAccess(n,LACCs⟦AspSeqElts AspSeqElt⟧(n,1),v,x-1)
               PAs⟦AspPrim⟧(n,0,v)
               STEPs⟦AspSeqElts AspSeqElt⟧(n,1,v)
               createSequenceAspect(n,x,v);
```

Figure 4.3: Implementation-level semantics: definition of sequence aspects

of the Arachne system, for example, in the presence of optimizations performed by the gcc compiler that may in-line function definitions but that conflict with the same function being used as jump targets for the rewriting strategies of the Arachne weaver.

In order to prove, at least in principle, the feasibility of the formal definition and correctness proof of the entire Arachne system (modulo the correctness of third-party tools such as the gcc compiler and the object file linker that are used as part of the Arachne tool chain) we have developed a complete formal semantics for Arachne's aspect language that allows reasoning over the Arachne compilation phase. This implementation-level semantics [3-DFL$^+$06, Fri05] is defined as a denotational semantics [Sto77, Sch86] whose valuation functions map Arachne aspect expressions to a domain of sequences of code generation functions. These code generation functions yield the exact C code that is generated, compiled and executed as part of the Arachne tool chain. They define symbolic references into the base code that are resolved at weave time to interface with Arachne's runtime rewriting libraries.

The well-definedness of the denotational semantics is straightforward to establish because of the rather simple domain structure of sequences of code generation functions: (i) the code generation functions that are produced may not be removed from such sequences afterwards, thus lower and upper bounds of different elements in the result domains are straightforward to define in terms of lattices over sequences; (ii) all definitions in the semantics that repeatedly add entries to sequences of code generation functions only use bounded, i.e., terminating, algorithms.

Figure 4.3 shows a typical excerpt of the implementation-level semantics that defines the semantics of sequence aspects as the valuation function **SA** yields the corresponding sequence of code generating functions based on the number of elements in the sequence (which is statically known). The result sequence consists of two groups of functions: a function producing initialization code and functions generating output for all steps in the sequence. Concretely, the initialization part (here performed by the function `defineAliasAccess`) sets up the reference to the base code mentioned above that, in particular, initializes the access to local aliases, and the three remaining expressions return the code for the sequence steps.

By successively replacing all non-terminals in the denotation of an aspect expression using the corresponding valuation functions, the implementation-level semantics allows to generate the exact code produced by the Arachne compiler (for an example of the rather lengthy resulting C code, see [Fri05], Sec. 4.2).

While this semantics provides a complete definition of Arachne's compilation scheme - and therefore contains definitions that are as detailed as its implementation itself - it does so by meeting two important characteristics. First, through a hierarchical design, it provides high-level abstract as well

as low-level detailed definitions. Second, it supports equational reasoning because of its pure functional nature. We have applied this last property, for instance, to prove that the Arachne implementation effectively observes the property of sequence aspects mentioned above in the context of the high-level semantics: that occurrences of join points matching the initial step of a sequence triggers creation of a new sequence instance without influencing the matching process governed by existing instances. Finally, we have derived performance characteristics of the Arachne implementation from the implementation-level semantics.

4.3 Conclusion and perspectives

In this section we have shown that expressive aspect languages based on the EAOP model provide appropriate means for the modularization of complex crosscutting concerns in sequential applications, both in the domain of component-based software engineering, which has been targeted by quite a large number of AOSD approaches, but also the domain of system-level applications, for which support using AOP has not yet been deeply explored.

Software components, which currently are the main structuring mechanism for large-scale applications, should benefit much from AOP, because of its prevalence of crosscutting functionalities. As we have discussed most approaches only address the manipulation of interfaces of existing component models, in particular of industrial component platforms, without explicit support for composition properties. In contrast, we have advocated an approach to AOP for components that enables composition properties to be investigated formally and partially be ensured by construction even in presence of component-modifying aspects. Technically, this has been realized by relying on protocols to make explicit in interfaces properties of the semantics of components. Aspects are then used to modify these protocols. This permits to investigate the influence of aspects on component properties as well as component-specific properties of aspect weaving. Finally, as we have shown that such an aspect system can be smoothly integrated in industrial-strength component platforms.

The second class of modularization problems we have discussed, system-level applications implemented using the C language, poses different challenges than concerns for software components. First, dynamic weaving of aspects into executing native code imposes strong restrictions as to the base language entities that can be referred to in aspects. Second, execution speed is crucial, which precludes the usage of interpretative execution techniques and, generally, requires aspect weaving and execution to be implemented with very small overhead. We have developed an aspect language for C featuring finite-state based pointcuts and aspects, devised a corresponding implementation with small overhead, and defined how to treat the resulting aspect system in a formal way.

Perspectives. This work offers a number of perspectives. It provides means, in particular, to address a general problem of software components that is particular critical for the modularization of crosscutting functionalities: striking a compromise between black-box and white-box access of software components. While software components are most frequently be defined as black-box encapsulation entities, the need for white-box or gray-box software composition mechanisms is well established [Aßm03]. However, the unstructured access of component implementation, in particular using aspects, defeats the *raison d'être* of components by breaking fundamental encapsulation properties. Aspects over component protocols essentially provide a black-box model but can naturally achieve gray-box like behavior by representing and manipulating additional information about the implementation of a component on the protocol level. More prospectively, this approach promises to contribute to a general integration of encapsulation mechanisms, including software components

but also modules, by generalizing the currently emerging proposals, e.g., Aldrich's notion of open modules [Ald05]. This lead of further work is discussed in more detail in Sec. 6.

As to the handling of system-level applications, our results prove that expressive aspect mechanisms can be used even in restrictive execution contexts. The Arachne aspect language currently still is, however, subject to several restrictions, e.g., concerning access to structured and local data that should be lifted. Our work has already fueled corresponding research, for instance on the inclusion of pointcuts enabling access to C-style records by Yanagisawa et al [YKCI06]. A second important lead of future work consists on the application of the techniques we have developed for C, an imperative language, for object-oriented languages used for system-level programming, in particular C++. While we have shown how the Arachne aspect model (regular aspects including) can be transposed into C++ [5-FSS$^+$05], much work remains to be done to fully explore which language mechanisms involving OO abstractions, such as late binding in the presence of C++'s multiple inheritance, can be reasonably applied as part of a strategy of dynamic weaving into native code. Finally, the potential synergies between AO languages defined based on source-to-source weaving of system-level applications [CKFS01, SGSP02, 5-ÅLS$^+$03] are another prime target for future efforts. Hybrid approaches that combine aspect weaving at compile and runtime promise, in particular, to be useful to lift the restrictions of runtime weaving mentioned above.

Chapter 5

Aspects for explicit distributed and concurrent programming

Distributed (and to a minor degree concurrent) applications should constitute a prime target for AOP methods: they are typically large, abound of crosscutting concerns and represent a large share of all practically relevant applications. In fact, two of the very first aspect languages, COOL and RIDL, developed by Christa Videira Lopes [Lop97, KLo97], respectively provided means for the modularization of mutual exclusion of concurrent activities and data transfer in distributed applications. Her work on explicitly distributed and concurrent aspects has been followed up only by very few approaches, most notably by the remote pointcuts for distributed AO programming introduced by Nishizawa et al. [NCT04].

In contrast, there is a large number of approaches that apply AOP techniques to distributed programming by using sequential aspect languages to manipulate platforms for distributed application development. Our approach of AOP over components with explicit protocols described in Sec. 4.1 belongs to this category, because distribution issues are left implicit (although component identities in protocols can be used to model, e.g., distribution domains). This is also the case for the numerous academic approaches (e.g., [CC04, DEM02]) as well as AOP frameworks, such as Spring AOP [Spr] and JBoss AOP [JBoa], for the modularization of functionalities in EJB-based distributed applications. Finally, the situation is very similar as far as concurrency is concerned: sequential aspect languages are overwhelmingly used to manipulate frameworks for concurrency, as recently exemplified by Cunha et al.'s [CSM06] who use AspectJ to improve the modularity of concurrent algorithms implemented using the concurrency library of Java5.

These approaches fall short with respect to the two main characteristics that guide the present study: declarativeness of aspect definitions and support for reasoning about properties of aspects. They are not declarative mainly because they do not allow to directly express essential relationship between distributed and concurrent entities. Spring AOP and JBoss AOP, for instance, follow AspectJ's atomic aspect model and thus do not allow, for instance, to directly express relationship between functionalities involving clients and servers but require them to be broken down in distinct aspect definitions manipulating both separately. It is well-known that such aspect systems that do not allow to quantify over different distributed entities in a distributed systems do not allow to concisely express distributed systems [SLB02]. Furthermore, as in the sequential case, such atomic aspect definitions require the use of non-local state, thus severely hindering reasoning about aspects. The need for explicit and more declarative pointcut and aspect definitions for distributed programming is therefore widely acknowledged. However, this problem has only very partially been addressed: in the context of 3-tier

architectures, for example, almost no support on the aspect language support has been proposed, a notable exception being Cohen et al.'s "tier-cutting pointcuts" [CG04] that introduce a limited from of quantification involving events at the client and server side.

Our approach to address these deficiencies harnesses two characteristics. First, explicit representations of relationships among distributed and concurrent entities on the pointcut level through finite-state based pointcuts. Second, the use of domain-specific abstractions on the pointcut and advice levels to reference and manipulate distributed and concurrent systems. Regarding distributed programming, the language and system of Aspects With Explicit Distribution (AWED) enables references to remote events in pointcuts and the synchronous or asynchronous remote execution of code as part of advice. As to concurrency, our recent results on Concurrent Event-based AOP (CEAOP) allow pointcuts to be defined in terms of multi-party rendez-vous synchronization *à la* CSP [Hoa85] and enable synchronization of aspects and base programs using explicit synchronization operators over (parts of) advice. Both approaches therefore allow non-sequential programs to be defined in a much more concise and declarative manner than with sequential AOP approaches. The AWED system furthermore has been implemented using a state-of-the-art Java based weaving and execution infrastructure yielding execution overhead close to a comparable hand-written, optimized RMI-based implementation. The CEAOP language, on the other hand, supports the verification of properties of coordinated concurrent aspects, for instance absence of deadlock, using model checking techniques.

The approach to distributed programming presented in this chapter has been developed in the context of the PhD of Luis Daniel Benavides Navarro I am supervising and Wim Vanderperren from SSEL group at Vrije Universiteit Brussel. The notion of concurrent aspects is the fruit of a cooperation with my colleagues Jacques Noyé and Rémi Douence from EMN.

In the following we give a detailed account of the AWED aspect language and system in Sec. 5.1. We describe the CEAOP approach more succinctly in Sec. 5.2 by briefly presenting its main characteristics and discussing synergies between aspects for explicitly concurrent and distributed programs.

5.1 Aspects with explicit distribution

Distributed programming poses several new challenges for AOP compared to sequential programming. Crosscutting concerns depend on execution events occurring as part of the execution of different distributed distributed entities (tasks, machines, cluster in LANs, subsystems of grids, etc.). Their modularization typically involves the execution of actions on several machines at once. Aspects have to manage distributed state, in particular, when state has to be passed from machines on which execution events have been matched to machines where actions have to be executed.

These challenges translate into different distribution-specific requirements for abstractions on the aspect language level:

- Pointcuts must be able to denote remote execution events.
- Advice should support triggering of remote actions.
- Aspects ought to provide mechanisms to manage internal state that may be shared or distributed. Furthermore, they should support different parameter passing modes for state of the base program and aspect-internal state that is referred to in pointcuts and used in advice.

The resulting aspect language should, in particular, be expressive enough to allow abstraction from auxiliary entities of distributed systems, such as proxies, in favor of direct manipulation of the corresponding remote entities.

Aspect languages for distribution also require specific execution support. Most fundamentally, runtime support for aspects in distributed systems need mechanisms to propagate remote execution events between sets of sites, thus typically favoring multicast-based communication instead of point-to-point communication. Second, aspect mechanisms often require aspect-specific information to be passed between distributed entities that cannot be transmitted transparently using implementation-level abstractions in existing platforms for distributed programming. This is the case, for instance, in the case of call stack information that has to be passed between machines in order to implement a distributed cflow pointcut. Third, some aspect mechanisms, such as advice execution on remote hosts, calls for advanced execution mechanisms, such as code mobility rather than standard remote method invocation. Code mobility is, however, not (or only rudimentarily) supported on common implementation platforms. Finally, it is unclear how distributed aspect runtime systems can be optimized, in particular whether optimization techniques for aspects developed in the sequential setting carry over to, or are even meaningful in, the distributed case.

The approach of aspects with explicit distribution (AWED) that we have developed provides a first set of solutions to these language-level and implementation-level issues and procures evidence that the resulting approach to the modularization of distributed systems effectively enables the concise definition of distributed crosscutting concerns. In the following we give an overview of the AWED aspect language and discuss how we tackled the implementation challenges mentioned above.

5.1.1 Language

The AWED language [5-BNSV+06a, 5-BNSVV06] directly addresses the three requirements for aspect languages for distributed programs introduced above.

Pointcuts may refer to remote execution events in a variety of ways, most basically by restricting matches of joinpoints to individual or groups of hosts. Host groups can be manipulated as first-class entities on the language level. Moreover, control-flow dependent and finite-state based pointcuts may match joinpoints occurring on different hosts.

Advice can be executed on (groups of) remote hosts. If an advice is to be executed on multiple hosts, the order of execution on these hosts can be specified explicitly. Advice can be defined to execute in a synchronous as well as asynchronous manner to the base execution. The synchronization mode also governs the execution of different advice that apply at the same execution event. Data dependencies between advice and the base program (or among advice that are simultaneously applied) are managed following a by-need synchronization policy realized using futures [HJ85, PSH04]. Advice is also used to dynamically register hosts with or remove hosts from host groups.

Figure 5.1 illustrates the main features of the AWED pointcut and advice language. Remote pointcuts may relate execution events on different distributed entities as illustrated by the solid directed path. Advice execution may be flexibly defined from the perspective of the (grayed) local host, i.e., the host on which the aspect is located (and where matching of the pointcut has been started): as illustrated by the dashed arrows advice may be executed on the local host, the host where pointcut matching has started, specific hosts (identified, e.g., by their IP address) or groups of hosts.

Aspects, finally, can be distributed through specific deployment specifications that allow to define, in particular, their state sharing properties. Aspects may be deployed — that is, their local state allocated — on single machines or globally within a network. Aspect state is by default encapsulated, it can, however, be explicitly shared among explicitly defined sets of instances of an aspect that are located on specific or groups of hosts.

Example: As a simple example illustrating the use of AWED, let us consider how to resolve

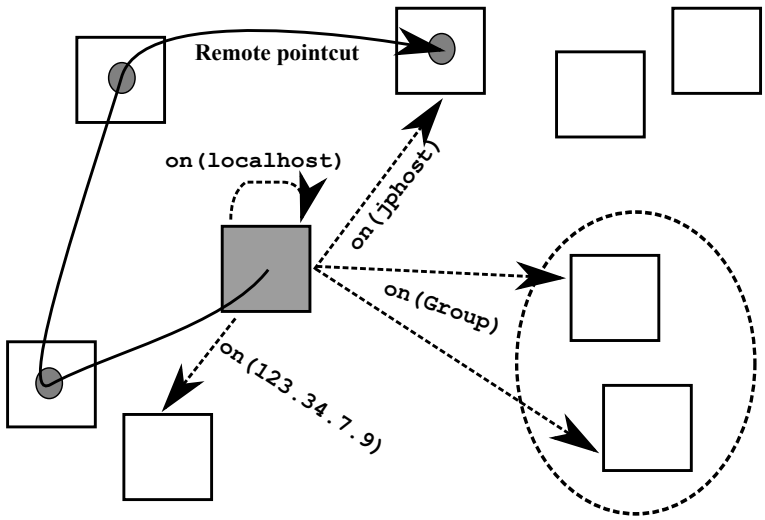


Figure 5.1: Remote pointcuts and advice

```
all aspect {
    pointcut distribution(Facade f):
        target(f) && call(* *(..)) && !host(ServerAddr)
            && on(ServerAddr);

    syncex Object around(Facade f): distribution(f) {
        return proceed(f.getInstance());
    }
}
```

Figure 5.2: Distribution as an aspect

the basic problem that has been identified *e.g.* by Soares et al. [SLB02] as a main issue for distributed programming with sequential aspect systems: serving a request from a client, which is performed through a call to a facade object of the server on the client host by a server on another site in a distributed system. In AWED, this execution pattern can simply be implemented by the aspect shown in Fig. 5.2, which is deployed on all sites in the system (as specified by the keyword `all`). The pointcut `distribution` matches all calls to facade objects on hosts different from the server host (term `host`) and executes the corresponding advice on the server (term `on`)[1]. The advice is synchronously executed once the pointcut matches and executes the corresponding server method, which is passed the concrete facade of the server through which the call has been made (e.g., in order to distinguish different server interfaces).

In order to close this short overview of the distribution features of the AWED language, let us recall the example of replicated caching shown in Fig. 2.4 (see page 17). Besides a simple use of finite-state based sequences, it also illustrates the use of host groups to succinctly denote sets of hosts of interest: in the example all hosts belonging to the cache can be quantified upon using the group `cacheGroup`.

5.1.2 Implementation

The AWED system also provides first answers to the implementation issues raised by aspects in distributed systems that have been introduced above: support for matching of remote events, management of distributed state relevant to aspects, transparent passing of information for pointcut mechanisms, multicast communication, code mobility and optimization of aspect execution in distributed systems.

Instead of implementing AWED ex nihilo, its implementation has been based on JAsCo [SVJ03, JAs], a dynamic weaver for sequential AOP of Java-based applications. Apart from the obvious observation that distributed aspects subsume support for sequential aspects, this choice has been motivated by several features of the JAsCo system that were directly suitable for extension to the distributed case. Most importantly, execution events are reified in the JAsCo runtime system and passed via connectors (that may provide aspect composition functionality) to aspects. Connectors are stored in connector registries in order to enable dynamic addition/removal of connectors and aspects. Replicating this infrastructure on the nodes of a distributed system has proven to yield an efficient runtime infrastructure for distributed aspects. The AWED implementation, see [5-BNSV+06a, 5-BNSVV06, AWE], which is publicly available as a module of the standard JAsCo distribution, installs a connector registry on each host where an aspect is deployed and propagates joinpoints among them as needed. Second, JAsCo integrates support for sequential finite-state based pointcuts and aspects [VSCDF05]: the AWED implementation generalizes these to distributed finite-state based pointcuts.

Besides extending features of the original JAsCo system to the distributed case, the different issues specific to AOP for distribution have been realized within the AWED implementation by harnessing extension mechanisms of the Java platforms, by resorting to Java-based communication libraries or by means of AWED aspects themselves. Concretely, transparent information passing required for control-flow and sequence pointcuts has been implemented based on Java's mechanism of customized sockets by extending techniques proposed generally for distributed programming in Java [PSH04] and for AOP in particular [NCT04]. Multicast communication that is necessary for, for instance, the propagation of joinpoint representations to the hosts mentioned in different branches of a finite-state based pointcut and to advice that is executed on multiple hosts, is realized using the JGroups communica-

[1] Note that `on(h)` terms appear in pointcut declarations because it is semantically be defined as a predicate that restrict pointcut matches to contexts in which advice can be executed on `h`.

tion library [JGr]. Remote advice execution is realized by activation of deployed aspects that refer to remote state if necessary, essentially a form of weak code mobility [FPV98] that minimizes transfer of runtime state. Finally, two features are implemented using AWED aspects themselves: sharing of state among aspects and parameter passing between constituents of (e.g., finite-state based) pointcuts as well as between pointcuts and advice.

Performance considerations. The AWED system is implemented by specialized code for AO and distributed functionalities that makes extensive use of the JAsCo implementation, the standard Java platform and third-party extensions of the Java platform, such as the JGroups library. This raises the question how it compares to alternative implementation strategies with respect to execution efficiency. Generally, the efficiency of software systems typically is evaluated by benchmarking them against comparable systems, by macro-benchmarks (determining the overhead they induce in reasonably sized applications) and micro-benchmarks (determining the overhead of small execution patterns implemented with and without the technology of interest).

Since there is no system for aspects with features for explicit distribution whose set of features comes close to AWED, we have run microbenchmarks and compared AWED implementations to functionally and distribution-wise equivalent hand-optimized implementations using RMI only [9-BNSV+06b]. These evaluations have revealed a small but non-negligible performance overhead of a factor of 1.5–4 for AWED compared to a plain Java implementation. However, a closer analysis of the performance bottleneck has shown that almost all of this overhead stems from use of the JGroups library for multicast communication: JGroups provides a communication model that by default is generic and can accommodate different transport protocols as well as different communication policies (such as resending lost messages until transfer succeeds). While we plan to exploit this feature (see the discussion of perspectives at the end of this chapter), an implementation of multicast communication that is specialized to our current needs would almost entirely eliminate overhead of AWED compared to a hand-optimized RMI implementation.

Finally, it is interesting to note that AWED is fully compatible with the sophisticated optimization mechanisms that are part of the JAsCo system. JAsCo uses two specialized optimizing modules for pointcut matching and the integration/removal of aspects during runtime [VS04]. The implementation of the AWED system has been carefully designed not to impair these optimizations, e.g., by ensuring that communication does not interfere with these optimizations.

5.2 Coordination-centric concurrency control using aspects

Much like distribution, concurrent execution of activities frequently crosscuts base functionality. Furthermore, concurrency control is often performed by following a number of well-defined concurrency patterns using well-known concurrency primitives [Lea96]. As in the distributed case, this argues for the use of a domain-specific aspect language for coordination of concurrent activities. Unlike distributed programming, concurrent programming is frequently modeled as coordination between a set of different activities of equal status. Such a coordination-centric perspective does not fit well with approaches to concurrent programming that employ aspects to introduce calls to concurrency primitives into a base application that is not concurrent.

> **Example:** Consider, for example, an e-commerce scenario in which updates to client views on the product database are introduced using aspects. Such updates may involve operations which influence or not an on-going transaction of a client. If they do they should probably

be performed in a synchronous manner to transactions so that clients are guaranteed to get updates as soon as possible, otherwise they may be performed concurrently to transactions. A coordination-centric approach permits to directly express that the aspect functionality, i.e., view updates, should be performed concurrently or not with the transactions, rather than express that low-level synchronization operations should be executed at specific execution events.

However, almost all existing approaches to AOP for concurrency control (including those by Lopes and Cunha cited in the introduction to this chapter) belong to this class. In contrast, a coordination-centric approach to concurrency control with aspects allows to directly express arbitrary functionality modularized using aspects with a base application as well as with other aspects. In the following we first detail the main features of such an approach to concurrent aspects and then sketch our first results on a corresponding aspect system, Concurrent EAOP.

5.2.1 Abstractions for AO concurrency control

Concurrency control through coordination of aspects and base programs most prominently relies on an appropriate notion of the entities of aspects that are *units of concurrency* and with respect to what events of the base execution these units can be synchronized with. We have explored coordination expressed directly terms of the abstractions provided by aspect languages. Pointcuts may be matched concurrently, e.g., independently during execution of concurrently-executing threads of the base program. AspectJ-style advice provides useful abstractions for synchronization of base programs and aspects: after advice, for instance, may be executed in sequence or concurrently with the base execution following the joinpoint whose match triggers the execution of the advice. Around advice allows for more flexible synchronization patterns because parts of around advice may be synchronized independently. Before advice, in contrast, can only be executed in a sequential manner before the matching joinpoint.

Once a well-defined notion of entities of aspects that can be synchronized has been defined, existing formalisms for concurrency, for example one of the numerous process calculi for concurrency, can be employed as a basis for language support to coordinate the concurrent execution of these entities and the base program. The coordination language for concurrent entities must be complemented by a corresponding weaver definition. This proceeding promises to enable formal reasoning about concurrent aspects by harnessing properties of the underlying concurrency formalism, provided that there is a suitable formal weaver definition.

Since such language support expresses coordination between entities explicitly, it is possible to define composition operators that provide abstractions of concurrency patterns on top of basic AO concurrency control mechanisms. Such composition operators can be used, in particular, to define the synchronization of the concurrent entities of several advice that apply at the same joinpoint. This way, composition operators address, in particular, the problem of interactions of concurrent aspects.

Furthermore, such a notion of concurrent system can be implemented using concurrency abstractions of mainstream programming languages, because it relies only on simple synchronization operations to be weaved around the entities of concurrent execution.

Finally, let us note that such a model instantiates the EAOP model as introduced in Sec. 2.4 in that it supports expressive aspects at the language level, reasoning about properties of AO programs and mechanisms for explicit aspect composition.

5.2.2 Concurrent Event-based AOP

The model of Concurrent Event-based AOP (CEAOP) [5-DLBNS06, CEA], which we have recently proposed, realizes a coordination-centric AO approach for concurrency in the sense above.

Units of concurrency. Units of concurrency are defined as outlined above: pointcuts can be matched on different threads simultaneously and parts of advice can be synchronized with the base execution and one another.

Before and after advice can be synchronized as discussed above. The most interesting case, around advice, deserves further explanation. Around advice can call the base functionality whose matching has triggered the advice using the special method `proceed`. Advice that calls `proceed` is structured in three execution segments: the segment before the call to `proceed`, the execution of the base functionality represented by the call itself and the segment after the call. These three segments give rise to different quite natural coordination strategies: the before segment has to be executed before the matched base functionality is executed, but the after segment may be executed synchronously or asynchronously with the base execution following the matched joinpoint.

In the case of different advice that apply at the same joinpoint, richer coordination strategies can be defined. In sequential languages following the AspectJ model, several advice are totally ordered and the before and after segments of around advice are executed in a nested fashion. In the concurrent case before (after) segments of different advice may be executed in a concurrent or sequential manner.

Around advice in AspectJ can call base methods multiple times using calls to the special method `proceed`. The corresponding segments before, between and after calls to `proceed` are potential concurrent entities. However, there are two technical difficulties. First, segments between calls to `proceed` cannot be unambiguously mapped to base functionality with which it should be executed in a sequential or concurrent manner. Second, how should around advice be coordinated with the base execution in the case of replacement advice, i.e., the advice-triggering base method is never called. These two issues have been addressed as follows: (i) CEAOP admits only one call to `proceed`; (ii) if the base functionality is not called, the keyword `skip` has to be used to mark the occurrence of the triggering base method relative to the around aspect.

Language: concurrent aspects and their composition Concurrent aspects in CEAOP are defined based on the formalism Finite State Processes (FSP) [MK99]. FSPs allow concurrent processes to be defined in terms of finite-state transition systems. CEAOP pointcuts correspond to such FSP automata, while advice is associated to individual transitions in this automata. The resulting core aspect language therefore is similar to the language of regular aspects, see Fig. 3.2, with the main exception that advice is structured in before and after segments that are separated by a call to either `proceed` or `skip` as described above.

Similar to regular aspects (cf. Sec. 3.3), CEAOP provides language support for aspect composition. In the case of CEAOP, these operators allow to coordinate different segments of advice that apply at the same execution event among another or with the execution of the base functionality. Such an operator can, for instance, execute all before segments of all advice applying at one execution event in a nested sequential fashion and execute all corresponding after segments concurrently. Formally, the composition operators are defined as FSP processes. They realize a wide variety of basic concurrency patterns that can be combined freely to implement sophisticated coordination strategies of concurrent activities.

Weaving of concurrent aspects and property support. Weaving of CEAOP aspects is formally defined by a transformation that maps aspects built from FSP expressions, inserted advice definitions and composition operators into FSP process definitions.

The weaver instruments the base program, aspect expressions and composition operators with synchronization events that correctly "implement" the coordination strategy defined by the concurrent aspect program. Basically, synchronization events are introduced to delimit the units of concurrency, i.e., segments of advice and base instructions corresponding to matched joinpoints. Composition operators are defined such that their synchronization events can be identified with those of aspects and the base program and thus apply the coordination strategy embodied by the composition operators to its argument aspects. Technically, the main issue for this approach is to define an instrumentation that introduces synchronization events that do not cause erroneous interactions between different aspects and composition operators, all of which are defined independently from each other. Such interactions are all too common in approaches like FSP where processes are synchronized that wait for events of the same name.

Since base programs, composition operators and the weaved programs are finite-state processes[2], existing analysis techniques and tools for this concurrency calculus can be used to formally establish properties over concurrent AO programs. The model checker Labelled Transition System Analyzer (LTSA), in particular, allows to verify FSPs safety as well as some lifeness and deadlock freeness properties.

5.3 Conclusion and perspectives

In this chapter we have motivated and introduced language support for the modularization of crosscutting concerns for distributed and concurrent programming. In both cases the resulting aspect models enabled much more declarative aspect definitions than with existing aspect systems because of the integration of non-atomic aspects and the definition of aspects in terms of domain-specific abstractions. Furthermore, both models provide explicit support for the composition of aspects. Finally, we have shown that aspects with mechanisms for explicit distribution can be reasonably implemented using standard infrastructure for distributed systems. Our approach to concurrent aspects allows formal reasoning about correctness properties, which are especially important for concurrent applications.

Perspectives. The work described in this chapter opens up a large number of new perspectives. A comprehensive "theory" of aspects for distribution and concurrency probably is one of the key issues not only in AOP but for software engineering in general: we will discuss this issue in some detail in Section 6.

Since concurrency issues obviously are an integral part of distributed applications, integration a model of concurrent aspects into one for distributed programming is an interesting perspective. As discussed the AWED model already includes a rather rudimentary model for the concurrent execution of remote advice. This model could certainly be enriched by integrating elements of the CEAOP model. However, such an integration raises two important question. From a conceptual viewpoint, is is to be evaluated to what degree an integration is useful: the CEAOP model has been developed for complex concurrent applications, such as graphical user interfaces, that require sophisticated concurrent coordination Marrying aspects for distribution and concurrency. Concurrency in many distributed applications is, however, much simpler. Second, on the model level, the coordination of concurrent

[2]Recall that aspects are not FSPs because of interleaved actions.

activities belonging to different distributed entities has probably be addressed differently than concurrency within a distributed entity.

A currently still open question for distributed aspects is how their properties can be formally defined and reasoned about. A first reasonable approach is to start from an existing calculus for distributed programming and extend it with aspect-oriented abstractions in order to then investigate properties of DO distributed programs. In the case of AWED, Caromel and Henrio's ASP calculus [CH05] seems particular promising, because it provides support for many abstractions (e.g., multicast communication, groups of hosts) in distributed programs that are part of the AWED languages.

A major open question concerning concurrent aspects is their efficient implementation in mainstream programming languages. the CEAOP approach relies strongly on the notion of synchronization by multi-party rendez-vous. These are, however, notoriously difficult to implement using simpler concurrency abstractions. Furthermore, they do not scale with system size without specific provisions. While this is a principle problem if multi-party synchronization is used all over a system, a promising approach is to partition an application into parts that can use only simpler synchronization means among one another.

Chapter 6

Conclusion and perspectives

In this habilitation thesis we have explored expressive aspect language mechanisms for the modularization of large-scale applications. Based on a taxonomy of advanced features for aspect languages, we have shown that, we have provided a large body of evidence that explicit relationships between execution events on the level of aspect languages foster the precise formal definition of aspect languages and enable automatic reasoning about fundamental and essential properties of AO programs.

As we have shown, this approach that has initially been substantiated in the context of different formal frameworks can be harnessed to address real-world modularization problems through instantiations of a general model for expressive aspect languages that have been seamlessly integrated into mainstream programming environments. We have demonstrated the benefits of higher-level abstractions for aspects in the context of black-box component models and system-level programming, which permits only limited access to runtime information and in which execution speed is of prime importance.

Finally, we have shown that expressive aspect languages that involve domain-specific abstractions provide substantial benefits in the domains of distributed and concurrent programming, two of the most important application domains for AOP that until now have been neglected to a large extent.

Overall, expressive aspect languages therefore support the development of large-scale applications that is based on the concise formulation, analysis and enforcement of abstract properties even in the presence of intricate crosscutting concerns. They therefore promise to evolve into a central tool for the development of large-scale software systems from a large range of applications domains.

To conclude this thesis, let us briefly consider three more prospective perspectives of further work that promise to leverage the demonstrated benefits of expressive aspect languages in the context of three fundamental problems of today's software engineering: (i) how to reconcile modular development, in particular information hiding and encapsulation principles, with aspects, (ii) comprehensive pattern-based support for the modularization of distributed programs and (iii) aspects as a central tool for general model-driven application development.

Reconciling modular development and aspects

Modular structuring principles, such as information hiding and strong encapsulation of software artifacts, underlie most modern software development methods. The corresponding restrictive notions of access to software entities have proved useful to facilitate understanding of large software systems as well as their development, in particular, by allowing separate compilation and deployment of parts of large software systems.

Aspects are all about modularization, however most frequently only in a much more restricted sense than required by the two modularization principles above: aspects modularize crosscutting concerns only to that extent that they are represented by one well-defined software artifact (e.g., an AWED or AspectJ aspect). Aspects seem to contradict the modularization principles in that many approaches to AOSD have been designed with invasive access to software artifacts in mind. Furthermore, the so-called obliviousness property of aspects [FF04], which allows unanticipated weaving of aspects with base programs that have not been prepared for aspect weaving in any way, is in clear contradiction with the notion of access to modules that is performed solely through well-defined and explicit module interfaces. A nearly universal consensus has by now emerged within the AOSD community that obliviousness has to be discarded in this context[1].

A model reconciling aspects and traditional modules should allow to address questions such as to what extent information hiding and encapsulation properties are preserved and modified. Conversely, it should give information on how modules can be used to limit the effects aspects may have on a base program. While aspects over black-box component models as discussed in Sec. 4.1 give first answers to such questions, they are of limited value because aspects over black-box components fully respect, by construction, the encapsulation properties of the underlying component-based application and, moreover, reveal few insights on information hiding properties.

Until now only very few approaches have been proposed to reconcile aspects with modules in a richer sense. Aldrich [Ald05] has proposed an extension of ML-style modules that enables the formulation of restrictions on which interface functions may be advised by aspects. The resulting hybrid model unfortunately very strongly restricts the modularization opportunities by aspects. Some approaches, such as the aspectual collaborations by Lieberherr et al. [LOML01], provide means to better align aspects with the structure (in particular, the inheritance relations) of a given object-oriented program. The resulting method does, however, respect only weak encapsulation properties. Kiczales and Mezini [KM05] have shown how to annotate OO interfaces with additonal information on where pointcuts may apply. Their proposal has not (yet) developed into a systematic method and does not enjoy formal support for modular properties, though.

One of the most promising proposals to reconcile modules and aspects constitute specific interfaces that mediate between aspects and base programs. Such *aspectual interfaces* may express restrictions on aspects that are imposed by the modular structure of the base program but also restrictions or modifications to the modular structure of the base program that are required for aspect application. Property-based AOP as advocated in this thesis promises to be useful in order to define a flexible notion of interfaces in this sense. Especially, the aspect applicability conditions introduced in Sec. 3.2.2 seem to nicely match the requirements for aspectual interfaces. Furthermore, since these conditions can be used to enforce dynamic properties it is probable that they nicely complement a recent proposal for aspectual interfaces by Skotiniotis et al. [SPL06] that defines mediating interfaces in terms of the static structure of OO programs.

Pattern-based distributed programming

Patterns, more specifically design patterns [GHJV94] and corresponding implementation-level patterns, have proved to be a highly useful semi-formal resource in sequential programming for the development, understanding and documentation of software systems. Different studies have shown by now that aspects are beneficial to pattern-based methods in that they facilitate the implementation of patterns, see, e.g., [HK02].

[1] Obliviousness seems to be indispensable only in the context of aspects over legacy software but not if aspects are considered as an program construction method.

While distributed programming should potentially benefit much from a pattern-based approach for reasons similar to those in the sequential case, relatively few approaches exist in this domain, be they proposed by academia or by industry (see [KS02, AMC+03] for notable exceptions). Two characteristics of distributed programming contribute to this problem: (i) distributed communication and computation schemes are frequently of unregular nature so that simple regular patterns cannot be employed and (ii) current infrastructure does not allow to directly implement many patterns simply because the patterns involve distributed entities that have to be treated in different parts of a distributed application.

Expressive aspects with explicit mechanisms for distribution and concurrency as presented in Chap. 5 directly address these concerns: using the AWED approach, for instance, complex combination of pipelined computations that are interleaved with master-slave like execution patterns can easily be defined, even if they involve matching of execution event and execution of computations on different machines. Expressive aspects as advocated in this thesis should therefore be useful for a comprehensive approach to pattern-based distributed programming.

Generalizing support for model-driven engineering

Model-driven engineering (MDE) in the sense of application development through transformation between models of different abstraction levels (see Schmidt [Sch06] for a recent overview) — from abstract design models via language-level implementation models to execution models — is highly interesting as a potentially universal program development methodology. There are, however, two fundamental open issues for MDE as a universal development process: (i) it is unclear whether suitable hierarchized sets of abstractions can be found and (ii) the transformations between different levels have to be precisely defined and, to a reasonable degree, supported through automatic transformation methods.

Crosscutting functionalities constitute one of the main conceptual and technical problems in addressing these issues. Since crosscutting functionalities are relevant throughout all of such a development process, AOSD techniques should be valuable to improve MDE-based methods. Furthermore, besides the well-known benefits of aspects on the implementation level (as partially discussed in this thesis) are complemented by first clear results concerning benefits of AOSD techniques on the design and architecture level, see, e.g., Sullivan et al. [SGCH01]. However, few concrete representations suitable for MDE methods have been proposed and, as discussed in this thesis, there are only very few automatic techniques for the manipulation of AO programs.

Expressive aspects promise to provide a (partial) solution to this problem, at least on the architecture, language and implementation levels. Their higher level abstractions set them up as a useful intermediate representation of crosscutting functionalities between design and implementation. Furthermore, their better support for the analysis and enforcement of properties allows transformation between different models to be rigorously defined and supported by automatic tools. Finally, since they can be integrated in high-level specifications as well as low-level execution platforms, they should be an appropriate tool to ensure properties that relate different abstraction levels, such as tracing of design properties in implementations. This way, the application modifications to higher-level abstractions on lower levels should be facilitated.

Bibliography

Note to the reader. This bibliography is divided in *two parts*:

- References involving the author of the present thesis. The corresponding references are categorized according to publication type: their cite keys start with a digit that denotes the corresponding publication category. Within a category, references are ordered as is customary using alphabetical order as principal criterion.
- Articles by other authors. These references are not categorized and use standard cite keys. References are ordered as is customary using alphabetical order as principal criterion.

Publications by Mario Südholt (categorized)

— Book chapters —

[1-DFS04b] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Akşit et al. [ACEF04], chapter 18.

[1-NDS05] Jacques Noyé, Rémi Douence, and Mario Südholt. Composants et aspects. In Mourrad Oussalah, editor, *Composants : concepts, techniques et outils*, chapter 6. Vuibert, February 2005.

— Editorial activities —

[2-CJS06] Yvonne Coady, Hans A. Jacobsen, and Mario Südholt, editors. *Aspect-Oriented Programming for Systems Software and Middleware*. Springer Verlag, October 2006. Special issue of Transactions on AOSD.

— International journals —

[3-DFL+06] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. *Transactions on Aspect-Oriented Software Development*, 1(1), March 2006.

[3-DS01] Rémi Douence and Mario Südholt. A generic reification technique for object-oriented reflective languages. *The Journal of Higher-Order and Symbolic Computation*, 14(1), 2001.

[3-SDML+06] Marc Ségura-Devillechaise, Jean-Marc Menaud, Nicolas Loriant, Thomas Fritz, Ré mi Douence, and Mario Südholt. Dynamic adaptation of the Squid web cache with Arachne. *IEEE Software*, 23(1), 2006.

— National journals —

[4-CND+04] Pierre Cointe, Jacques Noyé, Rémi Douence, Thomas Ledoux, Jean-Marc Menaud, Gilles Muller, and Mario Südholt. Programmation post-objets : des langages d'aspects aux langages de composants. *RSTI L'Objet*, 10(4), 2004.

— International peer-reviewed events with proceedings —

[5-ÅLS+03] Rickard A. Åberg, Julia L. Lawall, Mario Südholt, Gilles Muller, and Anne-Fran çoise Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of Automated Software Engineering (ASE'03)*, pages 196–204. IEEE, 2003.

[5-BNSV+06a] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, B. De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'06)*. ACM Press, March 2006.

[5-BNSVV06] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, and Bart Verheecke. Modularization of distributed web services using awed. In *Proc. of the th Int. Conf. on Distributed Objects and Applications (DOA'06)*, LNCS. Springer Verlag, October 2006.

[5-DFL+05] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. In *Proceedings of the 4th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'05)*. ACM Press, March 2005.

[5-DFS02b] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*, pages 173–188. Springer-Verlag, October 2002.

[5-DFS04a] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'04)*. ACM Press, March 2004.

[5-DLBNS06] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In *Proc. of the 5th Int. Conf. on Generative Programming and Component Engineering (GPCE'06)*. ACM Press, October 2006.

[5-DMS01a] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In *Proc. of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 170–186. Springer Verlag, September 2001.

[5-FS02] Andrés Farías and Mario Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In Springer Verlag, editor, *International Symposium on Distributed Objects and Applications (DOA'02)*, volume 2519 of *LNCS*, pages 995–1006, 2002.

[5-FSS+05] Thomas Fritz, Marc Ségura, Mario Südholt, Egon Wuchner, and Jean-Marc Menaud. Automating adaptive image generation for medical devices using aspect-oriented programming. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'05)*, September 2005.

[5-HPS97] Anne A. Holzbacher, Martin Périn, and Mario Südholt. Modeling railway control systems using graph grammars, a case study. In *Proceeding of the Second Conference on Coordination Models, Landuages and Applications*, LNCS. Springer Verlag, 1997.

[5-MLMS04] Gilles Muller, Julia L. Lawall, Jean-Marc Menaud, and Mario Südholt. Constructing component-based extension interfaces in legacy systems code. In *Proc. of the 11th ACM SIGOPS European Workshop*. ACM Press, September 2004.

[5-Süd05] Mario Südholt. A model of components with non-regular protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, LNCS. Springer-Verlag, April 2005.

— National peer-reviewed events with proceedings —

[6-DS00b] Rémi Douence and Mario Südholt. Une technique générique de réification dans les langages à objets. In *6th Int. Maghrebian Conference on Computer Science*, 2000.

[6-DS03] Rémi Douence and Mario Südholt. Un modèle et un outil pour la programmation par aspects événementiels. In *Langages et Modèles à Objets (LMO)*, pages 105–117. Hermes, 2003.

— International peer-reviewed workshops —

[7-ÅLSM03] Rickard A. Åberg, Julia L. Lawall, Mario Südholt, and Gilles Muller. Evolving an OS kernel using temporal logic and aspect-oriented programming. In *Proc. 2nd Int. WS on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 7–12, March 2003.

[7-DMS01b] Rémi Douence, Olivier Motelet, and Mario Südholt. Sophisticated crosscuts for e-commerce. In *Int. Workshop on Advanced Separation of Concerns at ECOOP*, June 2001.

[7-DS00a] Rémi Douence and Mario Südholt. On the lightweight and selective introduction of reflective capabilities in applications. In *Int. Workshop on "Reflection and Meta-Level Architectures" at ECOOP*, 2000.

[7-FS98] Pascal Fradet and Mario Südholt. AOP: towards a generic framework using program transformation and analysis. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1998.

[7-FS99] Pascal Fradet and Mario Südholt. An aspect language for robust programming. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1999.

— Technical reports —

[9-BNSV+06b] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, B. De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. Research Report 5882, INRIA, March 2006.

[9-DFS02a] Rémi Douence, Pascal Fradet, and Mario Südholt. Detection and resolution of aspect interactions. Technical Report RR-4435, INRIA, April 2002.

[9-DS02] Rémi Douence and Mario Südholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical Report 02/11/INFO, École des mines de Nantes, December 2002. 2nd edition.

[9-FS04] Andrés Farías and Mario Südholt. Integrating protocol aspects with software components to address dependability concerns. Technical Report 04/6/INFO, École des Mines de Nantes, November 2004.

Publications by other authors (not categorized)

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In Richard P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*. ACM Press, 2005.

[ACEF04] Mehmet Akşit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley Professional, October 2004.

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, et al. Optimising aspectJ. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05), Chicago, IL, USA, June 12-15, 2005*, pages 117–128. ACM, 2005.

[Ald05] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *Proceedings of 19th European Conference on Object-Oriented Programming (ECOOP'05)*, LNCS. Springer Verlag, 2005.

[AM04] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing (STOC-04)*, pages 202–211. ACM Press, June 2004.

[AMC+03] Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., Mountain View, CA, USA, 2003.

[And01] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 187–209, 2001.

[AOS] AOSD-Europe home page. `http://aosd-europe.net`.

[AOS03] *Proceedings of the 2nd ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'03)*. ACM Press, March 2003.

[AOS04] *Proceedings of the 3rd ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'04)*. ACM Press, March 2004.

[Asp] AspectJ home page. `http://aspectj.org`.

[Aßm03] Uwe Aßmann. *Invasive Software Composition*. Springer Verlag, 2003.

[AWE] AWED home page. `http://www.emn.fr/x-info/awed`.

[BA01] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.

[Bir89] Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science*, volume 55 of *NATO ASI, F*, pages 151–216. Springer-Verlag, Berlin, Heidelberg, New York, 1989. International Summer School, Marktoberdorf, FRG.

[BJJR04] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μabc: A minimal aspect calculus. In *CONCUR 2004 - Concurrency Theory: 15th International Conference*, 2004.

[BL04] Noury Bouraqadi and Thomas Ledoux. Supporting AOP using reflection. In Akşit et al. [ACEF04], chapter 12.

[BMD02] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages with logic metaprogramming. In *1st Conf. Generative Programming and Component Engineering*, volume 2487 of *lncs*, pages 110–127, Berlin, 2002. Springer-Verlag.

[BMN+06] Johan Brichau, Mira Mezini, Jacques Noyé, et al. An initial metamodel for aspect-oriented programming languages. Technical Report D39, AOSD-Europe, 2006.

[CC04] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In AOSD04 [AOS04], pages 56–65.

[CD06] Maria Cibran and Maja D'Hondt. A slice of MDE with AOP: Transforming high-level business rules to aspects. In *Proceedings of the 9th International Conference on MoDELS/UML*, LNCS. Springer Verlag, October 2006.

[CEA] CEAOP home page. `http://www.emn.fr/x-info/eaop/ceaop.html`.

[CG04] Tal Cohen and Joseph (Yossi) Gil. AspectJ2EE = AOP + J2EE: Towards an aspect based, programmable and extensible middleware framework. In Martin Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *LNCS*. Springer-Verlag, 2004.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CH05] D. Caromel and L. Henrio. *A Theory of Distributed Objects - Asynchrony, Mobility, Groups, Components*. Springer Verlag, 2005.

[CK03] Yvonne Coady and Gregor Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In AOSD'03 [AOS03], pages 50–59.

[CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of Joint ESEC and FSE-9*, pages 88–98, September 2001.

[CL02a] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In Cytron and Leavens [CL02b], pages 33–44.

[CL02b] Ron Cytron and Gary T. Leavens, editors. *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, March 2002.

[CL06] Curtis Clifton and Gary T. Leavens. MiniMAO1: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 2006. Special issue on Foundations of AOP.

[CSM06] Carlos A. Cunha, João L. Sobral, and Miguel P. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *Proceedings of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'06)*. ACM Press, March 2006.

[DD99] Kris De Volder and Theo D'Hondt. Aspect-orientated logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference*, volume 1616 of *LNCS*, pages 250–272. Springer Verlag, 1999.

[DEM02] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *Proc. of the 1st international conference on Aspect-oriented software development*, pages 65 – 75. ACM Press, 2002.

[Dij74] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer Verlag, 1974. Published in 1982.

[DW06] Daniel S. Dantas and David Walker. Harmless advice. In *Proceedings of the 33th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-06)*, volume 41, 1 of *ACM SIGPLAN Notices*, pages 383–396, New York, NY, USA, January 2006. ACM, ACM Press.

[EAS] IST project 1999-14191 EASYCOMP (easy composition in future generation component systems). `http://www.easycomp.org`.

[Far03] Andrés Farías. *Un modèle de composants avec des protocoles explicites*. PhD thesis, Université de Nantes, December 2003.

[FF04] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Akşit et al. [ACEF04], chapter 2.

[FN03] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(1):3–27, 2003.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[Fra] Fractal home page. `http://fractal.objectweb.org`.

[Fri05] Thomas Fritz. An expressive aspect language with Arachne. Master's thesis, Ludwig-Maximilians-Universität München, April 2005.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[GK06] Max Goldman and Shmuel Katz. Modular generic verification of ltl properties for aspects. Proc. of 5th Int. WS on Foundations of Aspect-Oriented Languages (FOAL'06), March 2006.

[Gro02] Object Management Group. *CORBA Components*, June 2002. Version 3.0.

[Has] Haskell home page. `http://haskell.org`.

[HJ85] Robert H. Halstaed Jr. MultiLisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[HL95] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, February 1995.

[HM98] Graham Hutton and Erik Meijer. Functional pearl: Monadic parsing in haskell. *J. of Functional Prog.*, 8(4):437–444, July 1998.

[HNBA06] Wilke Havinga, István Nagy, Lodewijk Bergmans, and Mehmet Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In Robert E. Filman, editor, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006*, pages 214–225. ACM Press, 2006.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Eaglewood Cliffs, N.J., 1985.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[Inc03] Sun Microsystems Inc. Enterprise JavaBeans specification, version 2.1, November 2003.

[JAs] JAsCo home page. `http://ssel.vub.ac.be/jasco`.

[JBoa] JBoss AOP home page. `http://www.jboss.org/products/jbossaop`.

[jbob] JBoss Cache home page. `http://www.jboss.org/products/jbosscache`.

[JGr] JGroups home page. `http://www.jgroups.org`.

[JN94] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, pages 527–629. Oxford University Press, 1994.

[Kat06] Shmuel Katz. Aspect categories and classes of temporal properties. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 106–134. Springer Verlag, 2006.

[KHH+01] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, et al. An overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. of the 15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LLNCS*, pages 327–353. Springer Verlag, June 2001.

[Kic96] Gregor Kiczales. Aspect oriented programming. In *Proc. of the Int. Workshop on Composability Issues in Object-Orientation (CIOO'96) at ECOOP*, July 1996. Selected paper published by dpunkt press, Heidelberg, Germany.

[Kic01] Gregor Kiczales. Untangling code. *MIT Technology Review: Emerging Technologies That Will Change the World*, Jan/Feb., 2001.

[KLo97] Gregor Kiczales, John Lamping, and Anurag Mendhekar others. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th Europeen Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[KM05] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE'05)*, 2005.

[KS02] Ajay D. Kshemkalyani and Mukesh Singhal. Communication patterns in distributed computations. *J. Parallel Distrib. Comput*, 62(6):1104–1119, 2002.

[KS03] Shmuel Katz and Marcelo Sihman. Aspect validation using model checking. In Nachum Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 373–394. Springer Verlag, 2003.

[Läm02] Ralf Lämmel. A semantical approach to method-call interception. In *Proceedings of the 1st ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'02)*, pages 41–55. ACM Press, March 2002.

[Lea96] Doug Lea. *Concurrent Programming in Java: design principles and patterns*. The Java Series. Addison Wesley, 1996.

[LL04] Karl Lieberherr and David H. Lorenz. Coupling aspect-oriented and adaptive programming. In Akşit et al. [ACEF04], chapter 7.

[LLO03] David H. Lorenz, Karl Lieberherr, and Johan Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46(5):542–565, September 2003.

[LOML01] Karl Lieberherr, Johan Ovlinger, Mira Mezini, and David Lorenz. Modular programming with aspectual collaborations. Technical Report NU-CCS-2001-04, College of Computer Science, Northeastern University, Boston, MA, 2001.

[Lop97] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[Lop04] Christina Videira Lopes. AOP: A historical perspective (what's in a name?). In Akşit et al. [ACEF04], chapter 5.

[LPS05] Karl J. Lieberherr, Jeffrey Palm, and Ravi Sundaram. Expressiveness and complexity of crosscut languages. In Gary T. Leavens, Curtis Clifton, and Ralf Lämmel, editors, *Proc. of Foundations of Aspect-Oriented Languages (FOAL'05)*, March 2005.

[McI68] M.D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proceedings of the NATO Conference on Software Engineering*, pages 138–155, Garmish, Germany, October 1968. NATO Science Committee.

[MK99] Jeff Magee and Jeffrey Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.

[MK03a] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *First Asian Symposium on Programming Languages and Systems (APLAS'03)*, 2003.

[MK03b] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proc. of the 17th European Conf. on Object-Oriented Programming (ECOOP'03)*, pages 2–28. Springer-Verlag, July 2003.

[MKD02] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In Cytron and Leavens [CL02b], pages 17–26.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM Symposium on Operating System Principles*, pages 129–142, October 1997. Appeared in ACM Operating Systems Review volume 31, number 5.

[NCT04] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut - a language construct for distributed aop. In AOSD04 [AOS04].

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 28(10) of *ACM SigPlan Notices*, pages 1–15. ACM Press, October 1993.

[NS06] Dong H. Nguyen and Mario Südholt. VPA-based aspects: better support for AOP over protocols. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*. IEEE Press, September 2006.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association of Computing Machinery*, 15(12):1053–1058, December 1972.

[POM03] Roman Pichler, Klaus Ostermann, and Mira Mezini. On aspectualizing component models. *Software – Practice and Experience*, 33(10):957–974, August 2003.

[PSH04] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for java futures. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 206–223, New York, NY, USA, 2004. ACM Press.

[Pun99] F. Puntigam. Non-regular process types. In P. Amestoy et al., editors, *Proceedings of the 5th European Conference on Parallel Processing (Euro-Par'99)*, number 1685 in LNCS. Springer Verlag, September 1999.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *Transactions on Software Engineering*, 28(9), January 2002.

[RSB04] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, New York, NY, USA, 2004. ACM Press.

[RSC+96] S. Ravi, S. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[Sch86] David A. Schmidt. *Denotational Semantics - A Methodology for Language Development*. Allyn and Bacon, 1986.

[Sch06] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, February 2006.

[SDMML03] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia L. Lawall. Web cache prefetching as an aspect. In *Proc. 2nd Int. Conf. on Aspect-Oriented Software Development (AOSD'03)*. ACM Press, 2003.

[SGCH01] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of*

the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, pages 99–108, 2001.

[SGM02] Clemens Szyperski, Domiinik Gruntz, and Murer Murer. *Component Software - Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 2nd edition, 2002.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schroder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In James Noble and John Potter, editors, *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Conferences in Research and Practice in Information Technology, Sydney, Australia, 2002. ACS.

[SHU06] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Expressing different conceptual models of join point selections in aspect-oriented design. In Robert E. Filman, editor, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06,)*, pages 15–26. ACM Press, 2006.

[SK03] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, sep 2003.

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, New York, November 4–8 2002. ACM Press.

[SPL06] Therapon Skotiniotis, Jeffrey Palm, and Karl Lieberherr. Demeter interfaces: Adaptive programming without surprises. In *Proceedings of 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, Nantes, France, 2006.

[Spr] Spring AOP home page. `http://www.springframework.org`.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[SVJ03] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: An aspect-oriented approach tailored for component-based software development. In AOSD'03 [AOS03], pages 21–29.

[Szy98] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 1st edition, 1998.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE'99*, pages 107–119, Los Angeles CA, USA, 1999.

[VJ05] Bart Verheecke and Viviane Jonckers. Stateful aspects for conversational messaging with stateful web services. In *Proc. of the Int. Conference on Next Generation Web Services Practices (NWeSP'05)*. IEEE, 2005.

[VS04] Wim Vanderperren and Davy Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In Robert Filman, Michael Haupt, Katharina Mehner, and Mira Mezini, editors, *DAW: Dynamic Aspects Workshop*, pages 120–134, March 2004.

[VSCDF05] Wim Vanderperren, Davy Suvee, Maria Augustina Cibran, and Bruno De Fraine. Stateful aspects in JAsCo. In *Proc. of the 4th Int. Workshop on Software Composition (SC'05)*, volume 3628 of *LNCS*. Springer-Verlag, April 2005.

[WKD04] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):890–910, 2004.

[WV04] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159 – 169. ACM Press, 2004. also TR no. 2004-745-10, Uni. of Calgary, http://pages.cpsc.ucalgary.ca/ rwalker/old/papers/walker.2004a.pdf.

[WZL03] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 127–139. ACM, 2003.

[YKCI06] Yoshisato Yanagisawa, Kenichi Kourai, Shigeru Chiba, and Rei Ishikawa. A dynamic aspect-oriented system for os kernels. In *Proc. of the 5th Int. Conf. on Generative Programming and Component Engineering (GPCE'06)*. ACM Press, October 2006.

[YS97] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions of Programming Languages and Systems*, 19(2):292–333, March 1997.