THE UNIVERSITY OF CALGARY

Improving the Modularity of Context-Sensitive Concerns through the Use of Declarative Event Patterns

by

Kevin Douglas Viggers

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

September, 2005

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Improving the Modularity of Context-Sensitive Concerns through the Use of Declarative Event Patterns" submitted by Kevin Douglas Viggers in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Robert J. Walker, Supervisor
Department of Computer Science

Dr. John Aycock, Department Member
Department of Computer Science

Dr. Wessam M. Hassanein, Outside Member
Department of Electrical and Computer Engineering

Date

Supervisor: Dr. R.J. Walker

ABSTRACT

Software modules communicate with one another to form coherent software systems. In order to facilitate this interaction, knowledge of the *communication protocol* is split up and embedded in each participating module, making local reasoning about the protocol difficult. Modules themselves become collectively responsible for seeing that the appropriate sequence of messages transpires. With protocols scattered in this way, and tangled amongst the details that intrinsically belong inside modules, the traceability, comprehensibility, and maintainability of the *concern* they represent, and of the system as a whole, tend to suffer.

Declarative event patterns (DEPs) are a means to implement communication protocols between modules in a localized manner. DEPs describe sequences of events in the execution of a system and include the ability to recognize properly nested structures. They allow a developer to describe a protocol at a high level, without the need to express extraneous details. A developer can indicate that specific actions be taken when a given pattern occurs. Protocol patterns are automatically translated into the appropriate source code instrumentation and automaton for recognizing a given pattern.

Using a proof-of-concept extension to the AspectJ language that we developed, two case studies were conducted. Each compares the use of DEPs with other approaches when implementing protocols involving path specific customization and authentication policies. DEPs were found to improve the modularity of such concerns, making it easier for software developers to develop and change context-sensitive crosscutting concerns.

Table of Contents

List of Tables

List of Figures

Acknowledgement

Above all, I would like to thank Rob Walker for continually steering me back towards the point of it all. Having a supervisor so genuinely concerned for passing on the craft that is research has been an joy. You have nearly infinite wells of both knowledge and patience. Thank you for letting me tap into (and drown) in both of them.

I would also like to thank the people upstairs (Petri Varsa, Dave Dembeck), the people downstairs (Shafquat Mahmud, Mark McIntyre, Reid Holmes), and others for providing a much needed human element to graduate life.

Dedication

To Daria, my love and ladybug saviour.

And in memory of Elliott.

Chapter 1

Introduction

> The Japanese have a small word – *ma* – for "that which is in between" – perhaps the nearest English equivalent is *interstitial*. The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviours should be.
>
> —Alan Kay (on messaging, and the unintentional emphasis of objects in Smalltalk) [47]

Communication is more than just the exchange of messages. Whether it be between humans or the parts of software, the act of conveying meaning is a connection between participants that takes place in sequence, a continuous expression of ideas that are shaped from the combination of lexical elements, their relative positions to one another, and their occurrence in some larger discourse. The set of facts or circumstances that surround a communication act—*its context*—is instrumental in determining its interpretation. For example, take the following two sentences[1]:

"Please use the toilet, not the pool"

"Pool for members only"

If read individually, these sentences have quite reasonable interpretations, seemingly pertaining to pool-side signage. However, if read in sequence, the meaning of the second sentence is dramatically (and somewhat disturbingly) altered.

"Please use the toilet, not the pool. Pool for members only"

Interpreting the semantics of a communication act based on its *occurrence in context* is inherently a part of natural languages. Unfortunately, with the programming

[1] Credited to Charles Fillmore [83]

languages with which we build software systems today, altering the semantics of our programs based on the historic context of execution is anything but natural.

In software systems comprised of many modular parts, the flow of communication between the modules shapes the behaviour of the system as a whole. In the absence of some automatic means of interpreting (or re-interpreting) events in the *execution context* as they transpire at runtime, today's software developers make use of *protocols*. Protocols script interactions and exist to provide a clear and indisputable interpretation of the semantics of a communication act given what has transpired up until that point. They can be explicitly acknowledged, specified, and adhered to, as is often the case with protocols such as the File Transfer Protocol (FTP [68]), or implicitly assumed as is often the case with the interactions between individual classes or interfaces found in object-oriented software systems. Whether explicit or implicit, adherence to a protocol provides a kind of pre-defined semantic interpretation; if the protocol is followed, a particular context may be assumed.

A problem with realizing protocols in software systems is that each participating module becomes responsible for playing out its own part in the conversation; consequentially, partial knowledge of the larger communication is necessarily split up and dispersed amongst each participating module; this is termed *scattering*. This scattering of knowledge leads to the "disembodiment" of the protocol as a whole. The protocol also becomes tangled amongst the details that belong inside other modules; this is termed *tangling*. Scattering and tangling together make it difficult to locate, understand, and change the protocol, the affected modules, and the system overall [84].

To meet the shortcomings of protocol implementation we have devised an approach that permits the recognition of program execution contexts to be specified and implemented in a localized way, separate from the details of the participating modules. *Declarative event patterns* (DEPs) are a means to implement context-establishing protocols between modules while retaining the ability to locate the parts of a protocol

realization (traceability), understand the implementation of a protocol (comprehensibility), and more easily alter the protocol and its realization (evolvability). Contained within a modular unit of their own, DEPs describe sequences of events in the execution of a system; these sequences represent execution contexts of interest—properties of the context that must hold in order to enable the specific semantic interpretation of a given communication act.

In the remainder of this chapter we lay out the general concepts supporting and impeding the modularization and implementation of communication protocols in programming languages of today. We begin with a look at modularity in Section 1.1.1, and define some important terms we use throughout this thesis. Section 1.1.2 explains the ideas of *dominant decomposition* and *crosscutting concerns*—two problems seen in existing programming languages that limit our ability to evolve our software systems. The aspect-oriented decomposition, heralded as a possible solution to crosscutting concerns, is described in Section 1.1.3. There are still some types of concerns, such as protocol concerns, that cannot be made modular using aspect-orientation alone. We discuss these *context-sensitive* crosscutting concerns in Section 1.1.4. This missing support was first noted in an approach called *implicit context*; we describe implicit context in Section 1.1.5. In Section 1.1.6 we briefly explain *communication history*, a first attempt at addressing this lack of support for context-sensitive crosscutting concerns. Declarative event patterns, our solution for modularizing context-sensitive crosscutting concerns, are discussed in Section 1.2. We conclude with a statement of our thesis (Section 1.3), and an overview (Section 1.4) of the remaining chapters.

1.1 Background

Separating a problem into parts more palatable to the human mind—*separation of concerns*—has been applied to software, and essentially all other engineering disciplines since their inception. When it comes to software engineering, explicit consideration

of separation of concerns is usually attributed to Dijkstra [22, 44]. Over the years various programming language constructs and methodologies have emerged in the name of better separation of concerns. From subroutines, to structured programming, to modules, to classes, to components, programming languages are still very much evolving. The driving force behind this is the bare desire to represent ideas more abstractly, more clearly, and in a more separate manner. It is our nature as humans to decompose complex ideas into more manageable pieces. As software developers, we would prefer that each of these ideas be fully decomposed, with each subpart represented by one and only one corresponding software entity; however, we have yet to achieve this goal. Software concerns still end up intertwined and dependent on one another, discounting the malleability that one might afford of the term *software*.

1.1.1 Defining Modularity

The definition of concern is intentionally vague. Software encodes mental abstractions; a narrowing of the definition would impose restrictions on the mental representation we can conceive in software. A software concern is quite literally anything relating to a software system that we deem noteworthy in isolation.

Definition 1 (Software Concern)

A software concern is any particular goal, concept, or matter of interest of either functional or non-functional nature.

The goal of modularity is to represent concerns in isolation, to separate them from their operating context so that they may be considered and altered on their own, in a local way. Insulating concerns from their operating contexts means hiding the internal details of the module from the greater surroundings [66]. The names used to identify hidden data, or private behaviours are examples of what modules aim to encapsulate.

Modular approaches in languages today require modules to interact in a shared concrete context of operation. Therefore modules should provide well defined and simple interfaces in order to limit the extent to which other modules may depend on them. By keeping these interfaces simple, the less coupled modules become. Put simply, the less constrained a module is to a given context, the more modular it becomes.

There are similar terms that are often used synonymously with modularity such as localization, encapsulation, and abstraction; however, these refer to individual facets of the larger term.

Definition 2 (Localization)

The extent to which code representing a concern is grouped together. Localization does not necessarily imply insulation from context, or low coupling.

Definition 3 (Encapsulation)

The extent to which some code is insulated from its context. Encapsulation does not imply that the contained code is related in a cohesive manner, or a representation of a single concern.

Definition 4 (Abstraction)

The extent to which some (potentially dispersed) code embodies only the essential qualities of a concern. Abstraction does not imply that the code is local, or that it is encapsulated.

Together, these three terms can be used to define what we mean by module.

Definition 5 (Module)

A unit of software decomposition typically corresponding to a syntactic entity. We consider an ideal module as one that embodies a single concern in an abstract, local, *and* encapsulated *manner.*

Some examples of modules include functions and procedures, classes, objects, and components. When we speak of the *modularity* of a software system, we mean as follows.

Definition 6 (Modularity)

The degree to which a software system is divided into modules. The more a module is isolated from its context of operation the less coupled it is to other modules in the system, and the more modular a system becomes. A good decomposition should have modules communicate with one another only along simple well-defined interfaces of interaction—this is termed loose coupling.

Some of the key properties we strive for in software systems are achieved through modularity. In this thesis we are particularly interested in the *evolvability*, *traceability*, and *comprehensibility* of modules and the systems in which they are embedded.

Definition 7 (Evolvability)

The ease with which software can be changed to correct faults, to improve performance, to adapt to new environments, or to provide different functionality.

Definition 8 (Traceability)

The ability to locate those parts of a software implementation that correspond to a given concern.

Definition 9 (Comprehensibility)

How understandable some program or part of a program is semantically, and conceptually by humans.

It should be noted that these properties are relative and generally reflect the degree to which they can be achieved. For example, in this thesis we will often claim that a given approach leads to more evolvable software. The degree to which some approach improves on another when considering these properties is notoriously fuzzy. There

is no general quantitative way to compare and measure these properties; instead we rely on qualitative observations through case studies. To that end, in this thesis we build up a body of evidence to support our claims in the absence of any formal proof.

1.1.2 The Dominant Decomposition

Mainstream languages and modularization mechanisms suffer from a purported limitation that has been referred to as the *tyranny of the dominant decomposition* [84]: given a set of mechanisms for decomposing, or modularizing, a software system, and the decisions made during that decomposition, some of a program's concerns will end up cleanly modularized while others will not.

There may be concerns, just as important as those captured cleanly in the decomposition, that do not fit cleanly into the chosen modularization; as a result, the implementation of such concerns end up scattered across many modules, entangled with what would otherwise be well-encapsulated concerns.

It is becoming clear that certain combinations of concerns defy clean separation under any single set of decomposition mechanisms [81]. In object-oriented decompositions for example, concerns as simple as logging the entry and exit of method calls fail to cleanly be contained in a single module. Other more significant functional concerns include synchronization, distribution, persistence, and authentication. Non-functional concerns, such as performance and other constraints similarly defy adequate modularization. These concerns *crosscut* the dominant decomposition [51].

Crosscutting concerns are largely responsible for impairing our ability to exercise change on our programs. Modularization aims to limit the cognitive burden placed on the developer by isolating a concern. When a concern does not fit well with the dominant decomposition in place, the developer is forced to scatter it throughout other modules, reducing the ability of others to comprehend the concern at hand, and increasing the risk of unintentionally breaking the existing implementations. The scattered code manifests into a tangle of dependencies amongst other software mod-

ules. These dependencies are a product of the supporting decomposition, and not necessarily inherent to the problem's *essential structure* [92, 89].

Acknowledgement of crosscutting concerns has lead to important advances not only in programming language separation of concerns, but also in other parts of the software development lifecycle: requirements [70, 11], design [18], and configuration [17]. This work has more recently been labelled under the research banner *aspect-oriented software development* (AOSD) [48, 3, 54, 59].

1.1.3 The Aspect-oriented Decomposition

The aspect-oriented decomposition seeks to decompose a software system as a collection of views, or perspectives on a common underlying solution to a problem. Views may or may not reference the same underlying elements or overlap one another. When perspectives refer to the same element, this corresponds to crosscutting in a purely one-dimensional modular decomposition.

The Unified Modeling Language (UML) [63], a common language for modelling object-oriented software, is aspect-oriented in the sense that a system may be comprised of a number of different diagrams, all of which focus on only particular details of the same underlying program. For example, when it comes to the patterns of interaction between objects, the UML provides, fittingly enough, interaction diagrams. Interaction diagrams come in two forms: sequence diagrams and collaboration diagrams. Both forms of interaction diagrams are founded on the same underlying information but convey or emphasize different *aspects* of the interaction. In particular, sequence diagrams show interaction arranged in a time sequence. The objects participating in the interaction are arranged by their lifelines, and the messages that are exchanged follow a particular order. We use sequence diagrams throughout this thesis to demonstrate particular patterns of interaction we hope to exploit to gain context (see Figure **??** for an example). Collaboration diagrams on the other hand, emphasize the relationships between objects, and the roles they play in the collaboration.

Representing the same interaction, these diagrams allow us to separate concerns, sequential and collaborative, during the *design* of a software system. Most mainstream languages lack the constructs at the programming level to support this separation at implementation. Aspect-oriented programming (AOP) is a notable exception.

AOP permits, at the programming level, the separation of crosscutting concerns in a modular fashion. Building on existing programming language constructs for decomposing systems into software, AOP provides a new modular unit called an *aspect*; aspects embody crosscutting concerns. Advanced compiler techniques are employed to compose, or *weave*, a coherent system out of the modules representing the base program, and those that represent crosscutting concerns over the base program [42].

Using AOP, one is able to write base modules that can be *oblivious* to the crosscutting concerns that may interact with it. Aspects *quantify* [33, 35] those points in the execution of a system where the crosscutting behaviour should interact with the base program—*join points* [51]. Additional or alternate behaviour may be applied at these points to alter the semantics of the program. Using this approach many crosscutting concerns can be localized, and encapsulated; however, there are still certain crosscutting concerns that fail to be cleanly abstracted with these mechanisms. This thesis attempts to extend the reach of aspects to separate these additional crosscutting concerns.

1.1.4 Context-Sensitive Crosscutting Concerns

Sometimes a concern's influence is short-lived, affecting the execution of a program based only on the occurrence of individual isolated events. This is the primary type of crosscutting concern that existing AOP approaches aim to address.

Other concerns tangle their way throughout the program, along the way establishing certain relationships between points (or state exposed at particular points) in a program's execution, and only influencing the program when a specific *execution context* holds. These contextually sensitive concerns have been termed *"stateful"*

crosscutting concerns [24, 25], because they often require the use of state-machine based approaches for monitoring their occurrence in the execution stream. Since protocols are used for establishing context, in this thesis we use the term *context-sensitive crosscutting concerns* synonymously with protocols.

1.1.5 Implicit Context

Despite the goals and claims of modularization constructs in use today, concrete context still seems to invade the insulating bounds of modules. This context can manifest itself as dependence limiting the modules' reuse. By delaying dependence on a concrete context for as long as possible, it is believed that the evolvability and reuse of a module will greatly improve [91].

Implicit context is a broadly-scoped software development approach for reconciling mismatches between modules in the presence of unanticipated evolution [91]. Fundamentally, implicit context is based on the idea of *contextual dispatch* between interacting modules. Using both static and dynamic adaptation constructs[2], contextual dispatch provides an implicit bridge between the local context surrounding individual modules, allowing these modules to interoperate despite expectations and assumptions on their operating context. An important facet of implicit context is in the way that it views modules; they are seen as pure abstraction lacking any concrete *referential* or *communicative* context. Instead, modules have a subjective, abstract, view of their surrounding context. At system composition time contextual dispatch acts to reify these abstract contexts.

Referential context Provides a world of objects to which one can refer. When using implicit context, modules are written (or treated as though they were written) under individual subjective referential contexts. The subjectiveness of the local context allows the module to focus on the *essential concern* which it addresses, foregoing

[2]The details of which are not relevant here, and which in fact are actively in flux.

the inclusion of any *extraneous embedded knowledge* (EEK) that might result from assumptions on a shared global context. In effect, an implicit context module is free to make any assumptions on *its* context, since its context is uniquely its own. References made from a module to its subjective context (for example, references to named types or instances in an object-oriented module), remain *abstractly bound*[3] to a module's local contextual view of the system. Through the mechanisms of contextual dispatch, a developer is able to specify mappings between the subjective referential contexts of interacting modules. These mappings are used at composition time to form a concrete referential context in which all modules may operate.

Communicative context Provides an abstract subjective view of the ongoing discourse in which a communication is situated. Often there will be times when the choice of referent or course of action is dependent on the execution context itself. Communicative context is involved with the semantic interpretation of what has occurred. Events situated in the past may be related, and historic values from previous communications may be carried forward for future use. Just as is the case with speakers of natural languages, communicative context helps to constrain or refine the interpretation of a communication act. Implicit context refers to, and supports, communicative context as *communication history* [92].

1.1.6 Communication History

Conceptually, the communication history of an executing program is a complete account of all communication acts that have occurred since the beginning of execution up until the current point of execution. Communication acts (events) can include, for example, the entry and exit from method calls and executions, the accessing and setting of fields, and object creations. Importantly, the referable context surrounding

[3]This seemingly contradictory term is used to emphasize that identifiers are bound to *elements* of a context that is not yet reified.

these events (such as the arguments passed to the method call, the target object, the calling object, and return values) are also retained in this history.

Communication history was first implemented and explored as part of IConJ, a prototype tool for implicit context. This realization had a number of shortcomings, particularly in terms of efficiency and mode of use. IConJ naively collected, as they transpired at runtime, all communication events in an indexed store which could be queried programmatically to explore patterns and retrieve state from past execution. While this complete record of events made this approach quite powerful, allowing for an open exploration of relationships between communication events, it also made it impractical. We address these shortcomings in this thesis, with an approach called *declarative event patterns*.

1.2 Declarative Event Patterns

Declarative event patterns are a practical realization of *communicative context* and, in conjunction with the features of AspectJ, provide an aspect-oriented approach to the clean separation of context-sensitive crosscutting concerns [93]. By describing the interactions of various entities at points in the execution of a program, and encapsulating this knowledge in a separated module (an aspect), DEPs endeavour to provide separation and modularization of protocol concerns while still retaining the nice properties of modularization: traceability, comprehensibility, and evolvability.

DEPs consist of declarative statements describing partial paths through the execution of a program, and resemble the production rule declarations of a context-free grammar. These rules specify a language of events; only the events that must occur in order to establish a requisite context. At runtime the actual events that transpire during an execution are validated against the specified patterns, dynamically inferring whether or not these contextual patterns hold. Important context such as argument values or references from past events may also be collected and exposed during this

process. Beyond simply confirming if a context is in place, DEPs permit a developer to alter the semantics of a program, and utilize exposed context, in a context-sensitive manner.

1.3 Thesis Statement

The thesis of this work is that declarative event patterns provide a practicable means for leveraging communicative context in software systems comprised of modular units, and that by using declarative event patterns, one can realize context-sensitive crosscutting concerns exhibiting improved evolvability, traceability, and comprehensibility as compared to other aspect-oriented implementation approaches.

1.4 Overview of Thesis

This chapter presented the problems with implementing protocols in existing modular software approaches and provided general background on context, modularity, and crosscutting concerns necessary to motivate the purpose of the research described in this thesis. The remaining chapters of this thesis are organized as follows. Chapter 2 provides further motivation by way of example; the inadequacies of existing approaches are demonstrated. Chapter 3 concretely describes declarative event patterns, our approach aimed at addressing the problems with existing approaches for realizing context-sensitive crosscutting concerns. Chapter 4 looks at the structure of URD, our prototype implementation supporting declarative event patterns. In Chapter 5, declarative event patterns are applied and validated against existing approaches. In Chapter 6, the novelty of declarative event patterns is shown through a comparison against related approaches. We discuss the applicability and future direction of declarative event patterns in Chapter 7. Finally, in Chapter 8 we conclude and summarize the contributions of our work.

Chapter 2

Motivation

Our goal in this chapter is to show that existing programming languages do little to directly support the establishment and leveraging of communicative context, and that such support is needed. We draw on a context-sensitive concern, a path-specific customization within an open-source mail client, to motivate the thesis of this work. Path-specific customization, as the term implies, involves applying certain custom behaviour only when a particular path through the program has been observed. In Section 2.1 we describe a customization example in detail, explaining its object-oriented implementation, and what is needed in order to realize our path-specific customization. Section 2.2 demonstrates an object-oriented realization of our customization and shows that it exhibits poor evolution characteristics. Aspect-oriented programming (AOP) techniques offer a potential solution to these inadequacies; Section 2.3 considers a concrete AspectJ [51] realization of our path-specific customization. Despite improvements over the object-oriented solution, we demonstrate that current aspect-oriented approaches do not address all the issues impeding the evolution of our customization concern—they lack support for context-sensitive crosscutting concerns.

2.1 Path-Specific Customization in Columba

Path-specific customization is one prevalent form of context sensitivity in object-oriented programs. Sometimes behaviours should only be exhibited by a program if certain events have previously been observed. Take for example the sequences of events required to successfully initialize an object, or for preparation of a system as a whole. Only after certain objects have been constructed, certain values assigned, certain relationships between objects established, can one count on a system to function as expected. If some step in the initialization process is neglected, then the assumed context, the operating environment expected by the interconnection of modules forming the system, cannot be trusted.

In the open-source mail client Columba[1], replying to a message residing in your **Sent** folder creates a message addressed to yourself and not to the original recipient of the original message you sent. This inconsiderate reply behaviour is prevalent in other mail clients as well, e.g., Mozilla Thunderbird[2], Microsoft Outlook[3], and SquirrelMail[4] all exhibit this behaviour. More often then not, if explicitly replying to a message that one has sent, one wishes to continue correspondence with the original recipient. To alter the behaviour of Columba to remedy this we can perform a path-specific customization. The base implementation of Columba's reply mechanism is detailed in Section 2.1.1. The necessary steps to establish the communicative context for our customization is discussed in Section 2.1.2.

2.1.1 The Columba Base Code

The Columba mail client is implemented in Java and follows standard object-oriented design. Replying to a selected message in a mailbox triggers the creation of a

[1] Columba is subject to the Mozilla Public License Version 1.1 (http://www.mozilla.org/MPL/)
[2] See http://www.mozilla.org/products/thunderbird/
[3] See http://office.microsoft.com/outlook/
[4] See http://www.squirrelmail.org/

`ReplyCommand` instance, and the invocation of its `execute` method. It is `ReplyCommand`'s job to assemble the header information for the new message in reply to the selected message and to prepare this new message for further composition. For our customization, we are interested in modifying this behaviour under certain contextual conditions. Columba makes use of the Template method design pattern [36], allowing subclasses of `ReplyCommand` to override certain behaviours of the `execute` method. This provides a protocol of interaction along the execution path, permitting variations in the path to occur through alternate behaviour contributed by subclasses. In Figure **??**, the Java source code for this `execute` method is listed. One template method, `initHeader` (called at line 19), is of significance to our example, and its default implementation is included in this figure.

FIGURE

The code listed in Figure **??** can be explained as follows. Lines 1 through 25 depict the `execute` method on `ReplyCommand`. Initially a new `ComposerModel` is constructed (line 3) and assigned to `ReplyCommand`'s `model` field. The model will hold message header information for the reply. Lines 6-9 promote a reference to the folder in which the selected messages reside into a `AbstractMessageFolder` local variable named `folder`. Line 11 obtains a set of unique identifiers—one for each of the selected messages for which the user wishes to reply. Lines 15 and 16 together create a private copy of the above retrieved information and assign it for use by the `ComposerModel` `model`. Things become more interesting when `initHeader` is invoked on line 19, and passed both the folder and message identifiers just retrieved. It is in the behaviour of the `initHeader` method, as depicted in lines 28 to 53, that our path-specific customization should take place.

The `initHeader` method is responsible for assigning to the `model` field the appropriate header information. Of primary importance, this header information includes, among other information, the sender and recipient of the reply. Despite the potential for multiple selected messages, `initHeader` chooses to operate only on the first of

these. In lines 30 and 31, the header information of the original selected message is obtained and used as the basis for a `BasicHeader` instance `rfcHeader`; the local `rfcHeader` is used throughout this method to access the original message's header information. Lines 37 to 40 determine to whom the reply message should be addressed. If the original message contained explicit `reply-to` header information, it is honoured. Otherwise, the new `to` header is populated from the existing `from` header. Columba provides an integrated contacts system that keeps track of the email addresses of those people the user has corresponded with; line 43 ensures that the `to` header has been recorded in this record. Line 44 updates the `model` field with the new `to` header information. The remainder of this method then updates the `model` field with additional header information pertaining to mailing lists (line 47), and from which of the user's accounts the message should be addressed (lines 50-52).

The `initHeader` method described is just one of many. The concrete subclasses of `ReplyCommand` override this method, making use of `getTo` and `getFrom` in ways different then the default implementation shows. We consider these alternate behaviours later in this chapter.

In this section we described the base implementation of the reply mechanism found in the Columba mail client. The presented implementation acts as a foundation to the remaining examples in this chapter.

2.1.2 Customization and Communicative Context

Communicative context helps to constrain the meaning of a communication act. For our customization, we must ultimately alter `getFrom` to instead assume the behaviour of `getTo`. We do not want to change the behaviour of the `getFrom` method outright; it is likely used elsewhere in the system and such a change could cause quite an impact. Instead, we want to be able to refine the behaviour of this method only when a certain communicative context holds. To achieve this we need to do two things:

1. Establish when the execution context is such that it is appropriate to apply a semantic change.
2. Leverage the execution context to conditionally apply semantic changes.

Establishing Context If one considers each execution path traced through a program as a string of events, then establishing communicative context amounts to ensuring that an observed execution path is a member of some language over these events. Certainly not every observable event in every possible execution of a program is significant. Establishing communicative context typically requires only a small subset of communication events from the concrete execution trace, such as the exiting or entering of certain method calls and executions. In our Columba customization for example, we are interested in just the events surrounding calls to the `getFrom` method, calls of the `initHeader` method, and calls to methods that can reveal which folder the message originated from, namely `getFolder`. And we are explicitly interested only in contexts (execution paths) where:

> [Calls to the `getFrom` method] occur within the dynamic extent of [calls of the `initHeader` method], but only if the original message for which a reply header is being constructed for resided in the [**Sent** folder].[5]

If we permit the above mentioned events to be written more concisely using subscripts to note the entry or exit of a method, and superscripts to note our interest in either the call or execution of the method, then this informal specification of context can be written as a simple pattern match on an abstracted execution trace involving only these events. For example, consider the pattern expressed as the GREP-style regular expression below.

$$(\texttt{initHeader}^{call}_{exit})?\ (\texttt{getFolder}^{call}_{exit})\ (\texttt{initHeader}^{call}_{entry})\ (\texttt{getFrom}^{call}_{entry})^{+}\ \$$$

[5] It should be noted that the above context specification is intended to be illustrative. It does not include the cases of replying to sent messages *not* residing in the sent folder. Nor does it cover cases when the message under question has a `reply-to` field set.

We are interested in execution traces that optionally (declared via the ?) begin with an event signalling the exit of a call to `initHeader`, followed by an event signalling the exit of a call to `getFolder` (and where the resulting folder is the **Sent** folder), followed by the entry event for a call to `initHeader`, followed by one or more (declared via the +) events noting the occurrence of calls to `getFrom`. The $ signifies that current point in execution. This expression matches only the contexts we are looking to customize, accounting for multiple occurrences of `getFrom` within any non-recursive execution of `initHeader`. How we go from a specification like the one above to actually establishing the execution context in code is largely up to the tools and techniques we use; this is the basic question against which the tools and techniques we explore in this thesis will be judged.

Leveraging Context Custom behaviours should only be applied when the established context holds. Sometimes a customization should be applied immediately following the establishment of context. Other times it is enough to simply know that at sometime in a program's past some pattern of events occurred. In the Columba customization, we decide to use the entry to calls to the `getFrom` method as part of establishing context, and apply our customization to the execution of the `getFrom` method, to instead execute `getTo`.

Just how to establish execution context, and leverage it to realize our customization depends on the languages and approaches we use. When translating a clear specification of the required context like the one above into an implementation using existing approaches like object orientation, the results are not always easy to comprehend or evolve.

2.2 Object-Orientation

With less complex path-specific customizations, such as our Columba example, one might be tempted to take a quick and dirty implementation strategy to realize the customization concern. In this section we demonstrate a more methodical approach, using popular and recognized design approaches; this is done in Section 2.2.1. Despite having carefully planned this implementation, certain fundamental problems remain; we discuss these in Section 2.2.2.

2.2.1 A Java approach

When moving from specification to Java realization, we designed our customization along these lines. First we established a formal finite state machine model of our context requirement (see Figure **??**). Next we translated that state machine into a Java class for monitoring context events. Finally we augmented the base code with additional behaviour to notify the monitor when significant events occurred at runtime, and to check the state of the monitor in order to conditionally apply our customizations.

State Machine With some careful analysis of our specification (Section 2.1.2) we derived the state machine equivalent to our contextual requirement as seen in Figure **??**. This state machine consists of four states: A, B, C, and D. The state machine begins in state A. A $\texttt{getFolder}^{call}_{exit}$ event followed in sequence by an $\texttt{initHeader}^{call}_{entry}$ event, followed by one or more $\texttt{getFrom}^{call}_{entry}$ events will arrive at accepting state D, by way of states B, and then C. While in state D, the required execution context holds. Receipt of an event that deviates from this sequence causes a transition back to state A regardless of the current state.

FIGURE

Monitor By combining elements of the State, Mediator and Observer patterns [36], we added to the base implementation a `Monitor` class to observe and maintain execution context as events in the execution transpire and to provide a single entity to query for contextual requirements. In our case, the `Monitor` was implemented to recognize the context specified by our formal state machine (Figure **??**). The `Monitor` class itself is shown in Figure **??**. Four methods corresponding to our four significant events are defined in the `Monitor` class. These are `observeGetSentFolderExit`, `observeInitHeaderEntry`, `observeInitHeaderExit`, and `observeGetFromEntry`. When an event occurs during execution, the corresponding method is intended to be invoked.

FIGURE

Instrumentation The monitor requires that the base code be *instrumented* at points where significant events occur. By instrumented we mean that additional code should be added at those points that will notify the monitor of an event's occurrence. Additionally, at points in the source code where customization is required, code must be added to make queries on the monitor, and to conditionally provide the customization behaviour itself.

Figure **??** lists an altered version of `execute` and `initHeader` methods. These interact with the `Monitor` to establish and leverage communicative context. The four methods defined on the `Monitor` class (`observeGetSentFolderExit`, `observeInitHeaderEntry`, `observeInitHeaderExit`, and `observeGetFromEntry`) drive the recognition of execution context, each of them considering the effect of their occurrence on the state of the recognizer. They are invoked throughout the source code listed in Figure **??** (see lines 10, 21, 23, and 44). On line 10, an additional check is made on the resulting folder to determine if it is the **Sent** folder. In Columba, the unnamed constant `104` is designated as the unique identifier for the **Sent** folder. The method `checkContext` is invoked on line 45 to query the monitor to determine if the context has been established. The interaction between the base code and the

`Monitor` can be visualized as a sequence diagram as seen in Figure **??**. This figure graphically shows the instrumentation (by way of `observe(..)` messages), and conditional checks against context (by way of the `checkContext(..)` messages). During runtime these instrumentation points announce events to the monitor, which provides the necessary recognition of context.

FIGURE

FIGURE

In this section we have shown just one object-oriented approach for realizing our path-specific customization. While other approaches certainly exist, this employs commonly recognized design patterns and may be considered representative of other solutions.

2.2.2 Critique

The object-oriented solution presented in the previous section manages to realize our customization concern, yet there are problems with it. Traceability, comprehensibility, and evolvablity are all important software engineering properties of a program; with these in mind, in this section we reveal this solution's flaws.

The most immediately visible problem with this solution is that, in order to report all significant events to the monitor, instrumentation must be scattered through the code. Our customization is relatively simple, and yet four different points in the default reply path needed to be touched. Code for checking the context and performing the customization was also required. In Figure **??** these changes have boxes drawn around them so that they are easily spotted. However, spotting these sorts of instrumentation points in a program is not always easy. In general, an entire program would need to be considered in order to manually locate and provide event-emitting instrumentation to drive a finite state machine. For instance, our example focused on one execution path, using only the default implementation of `initHeader`. We mentioned that `initHeader` acts as a template method, and is overridden by subclasses

of `ReplyCommand` to provide variations in how the message header information is initialized. If the current and future subclasses of `ReplyCommand` are taken into account, then event-emitting instrumentation must also be added to each of these separate extensions. It is possible that some of these instrumentation points could be overlooked or even mistakenly reported as another event type. Tarr *et al.* [84] have shown that this sort of scattering leads to problems understanding and evolving the customization. The presence of instrumentation in dispersed places in the code also detracts from the concern being addressed in those places. Localizing all those parts significant to a concern tends to improve its traceability, comprehensibility and changeability. We are unable to localize the event-emitting portion of our customization.

The way that context is established in the object-oriented solution is less than perfect. Even though the recognition of communicative context has been localized into the `Monitor` in this solution (see Figure **??**), it is still difficult to understand, change and trace back to the specification. We began with a declarative expression of those execution contexts of interest to our customization. Our resulting implementation does not only not resemble that specification, but is an implementation that attempts to follow a reinterpretation of that specification, namely the formal state machine. The state machine is able to capture the semantics of the contexts of interests succinctly, but it is difficult for a human to compare against our original patterns of events. That is, traceability is poor. And the connection between the finite state machine intermediate specification and the implementation ends up unclear. Even for our relatively simple expression of context, the implementation is complex. In order to gain an understanding of what contextual pattern the monitor aims to recognize, one must mentally reconstruct the state machine from a tangled set of states and transitions. That is, it is harder to understand than the expression of context we started out with.

If a change to the contextual specification were required, it would be difficult to perform on our implementation. Change tasks are error-prone when the developer

fails to understand the connection between the source code they are changing and the original concern. The evolvability of our object-oriented approach is limited in that the realization of the finite state machine is an obfuscation of the original context specification. In order to illustrate this, consider what would happen if we changed our specification from

$$(\texttt{initHeader}^{call}_{exit})?\ (\texttt{getFolder}^{call}_{exit})\ (\texttt{initHeader}^{call}_{entry})\ (\texttt{getFrom}^{call}_{entry})^{+}\ \$$$$

to

$$(\texttt{initHeader}^{call}_{exit})?\ (\texttt{initHeader}^{call}_{entry})\ (\texttt{getFrom}^{call}_{entry})^{+}\ \$$$$

That is, we remove the requirement that the customization be limited to the **Sent** folder. Now, whenever a reply occurs, we want our customization behaviour to occur. Whether this is a sound customization or not is not significant to the point. What is significant is that one would need to then reconstruct a state machine from the new pattern (or attempt to alter the existing one), and then re-translate that state machine into a monitor class (or attempt to alter the existing monitor class). Additionally, they would need to remove all instrumentation across the entire system pertaining to the absent event. The connection between the specification and the state machine is not immediately obvious. And the connection between the state machine and the code in the `Monitor` class itself is not clear. Attempting to evolve the existing `Monitor` to conform to a new state machine would be tedious and error-prone; transitions may be omitted, or possibly even made to the wrong state.

The realization of our path-specific customization is necessarily dispersed throughout the base implementation. Establishing context can be complex, even when that context of interest is not. This unclean disembodiment of our concern leads to poor traceability, comprehensibility, and evolvablility.

2.3 Aspect-Orientation

The customization of the message reply behaviour in Columba involves events in the execution of the program that are the result of code defined in different modules. An object-oriented implementation requires that each of these different modules be modified in some manner in order to realize this concern. Aspect-oriented programming (AOP) aims to improve the modularity of such crosscutting concerns. It is reasonable then to investigate whether AOP approaches can help us in this case. In Section 2.3.1 we look at AspectJ, an aspect-oriented extension to the Java programming language. We show that, while AspectJ does help to modularize our concern to a certain degree, the resulting implementation is still difficult to evolve, trace, and comprehend.

2.3.1 AspectJ

AspectJ considers a system to be comprised of a set of core concerns and a set of crosscutting concerns. Core concerns can be adequately represented in the base language (e.g., Java in the case of AspectJ). Crosscutting concerns by their very nature cut across the behavioural boundaries of the core concerns. Although a crosscutting concern (such as distribution, or persistence) can be represented in the base language alone, when this is done the resulting implementation of that concern becomes scattered across different parts of the system; the code is necessarily tangled amongst code implementing other concerns of the system. This makes it difficult to trace, to comprehend and to evolve not only the crosscutting concern but those core concerns in which it is tangled. AspectJ permits the separation and modularization of crosscutting concerns in an effort to regain these software engineering properties. In this section we look at the features of AspectJ relevant to our forthcoming aspect-oriented solution to the Columba customization. These are *aspects, advice, pointcuts,* and *context exposure.*

2.3.1.1 Aspects

AspectJ allows one to localize otherwise crosscutting behaviour into a single unit—an *aspect*. If and when a change is required to that crosscutting concern, the concern in its entirety is represented in just one place. The intention here is that if the code is located in a single place, it is in general easier to understand and to modify. Aspects resemble classes in Java, and may contain method-like declarations called *advice*.

2.3.1.2 Advice and Pointcuts

Advice implements the behaviour of a crosscutting concern. Advice is woven into the base code by the AspectJ compiler to execute before, after, or in place of declared sets of join points. The developer specifies these points using constructs called *pointcuts*; each piece of advice acts upon a pointcut. An extensive set of *primitive pointcuts* are supported for classifying individual points in the execution of a system such as method executions, field sets, or class initializations. Primitive pointcuts may be combined through Boolean conjunction (`&&`), disjunction (||) and negation (!) operators to build up more complex pointcut expressions.

In Figure **??** an aspect has been declared that represents a simple crosscutting concern. This concern involves observing entries to all methods named `initHeader` that are members of `ReplyCommand` or any of `ReplyCommand`'s subclasses. The figure contains a single aspect called `Observer`. Within this aspect one pointcut and one piece of advice are declared. The pointcut is named `observedMethods`. It describes the set of join points involving calls to a method called `initHeader` on the class `ReplyCommand` or any of its subclasses; the `initHeader` method must additionally have a result type of `void` and have an `AbstractMessageFolder` and `Object` array as formal parameters. Advice is declared to act on the `observedMethods` pointcut. The advice behaviour is declared to explicitly run just `before` the method is called. Note that this behaviour is equivalent to the manual instrumentation required to announce

events in the object-oriented solution (see Figure ??, line 21). Unlike the object-oriented solution, this piece of advice simultaneously addresses all implementations of `initHeader` conforming to the specified method signature. With one declaration, we have achieved the required event-emitting instrumentation for all of the different realizations of `initHeader`, in potentially many different modules across the entire system.

FIGURE

The AspectJ compiler identifies all the points in the base functionality of the system corresponding to the dynamic invocation of any of the matching `initHeader` methods in `ReplyCommand` or its subclasses; the additional code declared by the before advice is compiled into the base functionality so as to execute just before the occurrence of the identified points. The compilation phase that locates these points in the base program and incorporates into the program the behaviour specified by advice is called *weaving*; for this reason, aspect-oriented compilers are sometimes called *weavers*.

AspectJ's pointcut language provides a number of *wildcard* features to express a wide range of join points in a concise manner. The `+` symbol used in the example above is one such wildcard—it allows one to refer to a type or any of its subtypes in just a few characters. An ellipsis wildcard (`..`) can be used to under-specify Java packages and method parameter signatures. Kleene closure (`*`) can be used to partially specify Java identifiers. Complete coverage of AspectJ's pointcut language is available elsewhere [51, 50].

2.3.1.3 Context Exposure

Context exposure is a feature of AspectJ that allows one to retrieve and manipulate the state surrounding a join point. Pointcuts may be declared in such a way as to expose this state for use in advice. AspectJ defines primitive pointcuts that can be used to expose the arguments (`args`), target object (`target`) and current object (`this`) with

which a join point is associated. The return results of a method invocation, or an exception thrown may also be exposed using special variants of after advice.

The code listing in Figure ?? demonstrates an aspect exposing an `IFolder`, the result returned from an advised method call `getFolder`. The example contains a single aspect called `ExposeFolder`. Within this aspect one pointcut and one piece of advice are declared. The pointcut is named `callsToGetFolder`. This pointcut captures the set of of join points involving calls to method `getFolder` and which are defined on the class `FolderCommandReference`; `getFolder` must have a result type of `IFolder`. Additionally, a special after advice form is used to exposure return value; this is then bound to the advice's formal parameter `folder`, and used in the advice body to check if the folder is the **Sent** folder. This advice application should seem familiar to the manually implemented check seen in Figure ??, lines 6-11.

FIGURE

AspectJ (and other aspect-oriented approaches) allow one to avoid invasive changes to existing code. Instead, a module is altered implicitly by a modular aspect. The base code need not know about such modifications. Crosscutting behaviours are woven into the base code and implicitly executed when needed. This intuitively permits us to observe events necessary for establishing context without altering the points in the program where those events occur. Similarly, it allows us to alter points in the program where customization should occur, without those points being explicitly altered in code.

2.3.2 An AspectJ Approach

AspectJ alleviates the developer from having to invasively modify modules in order to instrument for each significant event. Instead, advice may be used to implicitly instrument the occurrences of these atomic events in the execution context. Similarly, advice may be applied at the required points to check if a given context holds. The task of monitoring context ends up being very much the same as it was with

the object-oriented solution. AspectJ does not explicitly support the recognition of communicative context. In this section we consider an aspect-oriented solution to our customization.

Localized Instrumentation and Monitoring The AspectJ solution amounts to a single AspectJ aspect named `ColumbaCustomization`, as seen in Figure **??**. This aspect operates over the base Columba implementation, advising event-emitting points in the program's execution, as well as points where custom behaviour should be applied. You can consider the events marked in Figure **??** to implicitly occur in the AOP solution. By contrasting the sequence diagram in Figure 2.1 with Figure **??**, this implicitness becomes visually obvious. The implicitness of the aspect-oriented solution is marked with gray lines.

FIGURE

In order to advise these points in the base code and provide the event-emitting or customized behaviour, we first declare three pointcuts: `initHeader` (line 3), `getFrom` (line 4), and `getFolder` (line 5). The `initHeader` pointcut specifies method call join points corresponding to methods named `initHeader` on `ReplyCommand` or any of its subclasses. The `getFrom` pointcut similarly captures call join points of methods named `getFrom` on `BasicHeader`. The `getFolder` pointcut permits the application of advice on join points corresponding to calls to methods named `getFolder` on the `FolderCommandReference` type.

With pointcuts declared, we next advise the event-emitting join points with a combination of before and after advice declarations. Lines 8-10 declare `before` advice that provides event-emitting behaviour that is to occur before calls to `initHeader` (those join points specified by the pointcut named `initHeader`, declared on line 3). Using the same pointcut, lines 11-12 provide event-emitting behaviour that is to occur `after` occurrences of calls to `initHeader`. Lines 14-16 declare before advice to emit events before calls to the method `getFrom`, as specified by the pointcut `getFrom` (line

4). Lines 17-21 use a special form of `after` advice to first expose the folder result returned from join points specified by the pointcut `getFolder`. The advice body consults the exposed `folder` value; if it is the **Sent** folder (equal to `104`), event-emitting code is executed.

In order to provide code to perform our customization, one final piece of advice is declared. On lines 24-27, we use `around` advice to provide behaviour that is to execute in place of the normal execution of the `getFrom` method declared on `BasicHeader`. We specify a pointcut in place in order to capture execution join points of the method `getFrom`[6]. Additionally, we also expose the current object, through the use of the context exposing primitive pointcut `this`. The advice formal `header` is assigned the value of the current `BasicHeader` as a result, and used in the advice body to instead invoke the behaviour of the `getTo` method on `BasicHeader`. One important and final point regarding this advice is that it will only be applied conditionally. The `if` pointcut is used in the advice declaration to limit the application of this advice to contexts where the current `state` in the finite state recognizer is equal to `D`. It is in this way that we achieve context sensitivity. The `state` variable, as well as the methods and other declarations for recognizing context are conceptually equivalent to those used in the object-oriented approach. The bodies of the event-driven methods have been omitted from this figure.

FIGURE

2.3.3 Critique

The AspectJ realization addresses some of the shortcomings of the object-oriented solution. However, AspectJ, and other aspect-oriented approaches, still lack explicit support for establishing communicative context in a manner that promotes easy evolution, comprehension and traceability.

[6]We could also have declared this as a named pointcut and referenced it instead of specifying it in-place

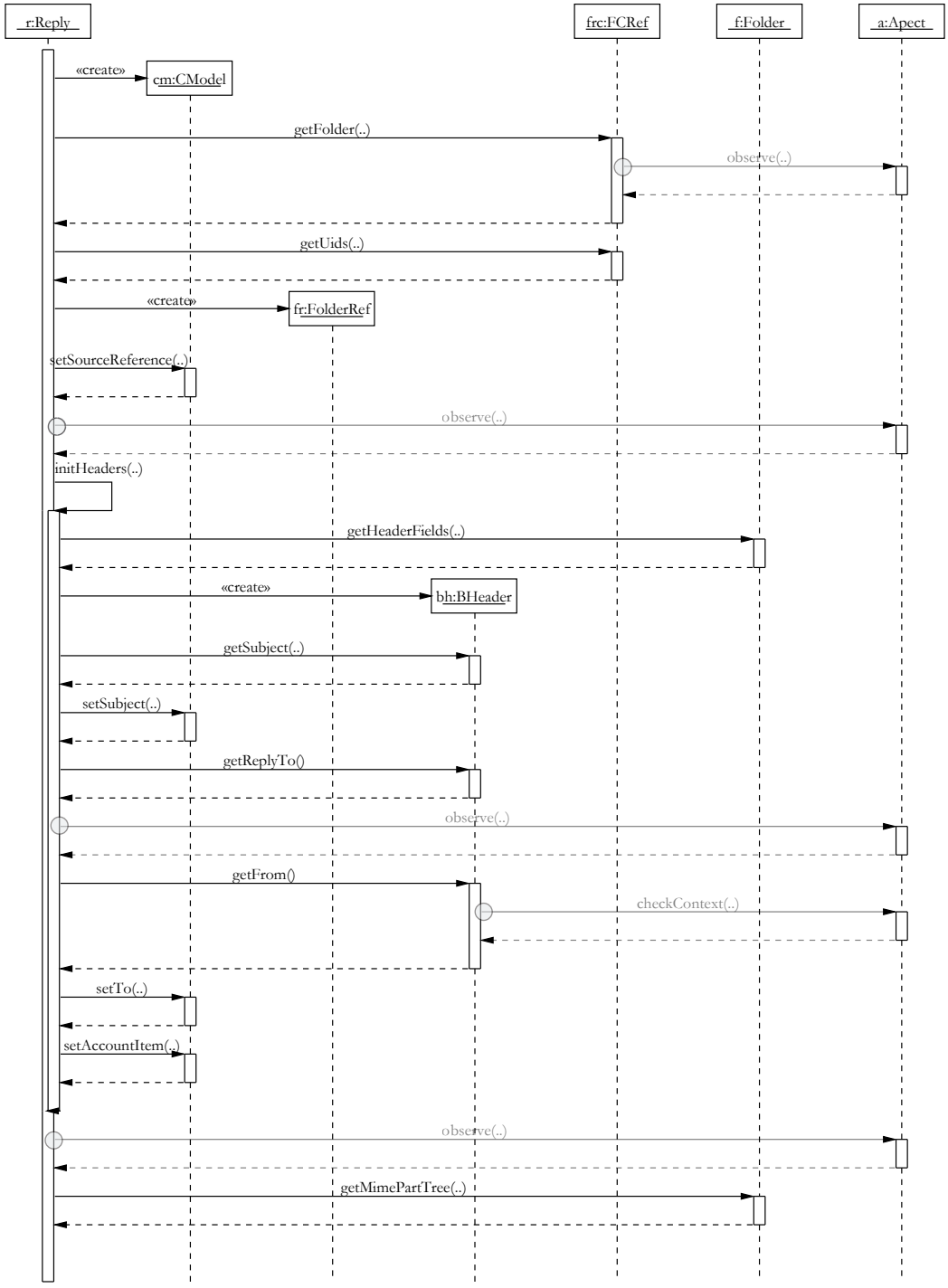

Figure 2.1. *AspectJ is used to monitor event occurrence*

The monitor object is now an aspect in this solution, and the explicit observation of events that were dominant in the object-oriented solution are implicit by way of our use of AspectJ [49]. However, making sense of and maintaining the relationship between these events still falls on the shoulders of the developer. The developer must still manually recognize the events of interest in order to piece together the context in a way equivalent to the object-oriented approach (Figure **??**).

The AspectJ solution provides a degree of improvement over the object-oriented solution we have presented. By allowing the announcement of events and the application of customization code to occur implicitly through AspectJ advice, the dependencies on the `Monitor` singleton are removed from the base code. This removed the scattering and tangling of this facet of the customization concern, localizing the concern in its entirety to a single aspect module (see Figure **??**). This is an important improvement over the object-oriented approach.

The aspect-oriented realization is more comprehensible than its object-oriented counterpart, but still exhibits complex and difficult to understand code. The inclusion of explicit declarative pointcuts (Figure **??**, lines 3-5) coupled with event-emitting advice (Figure **??**, lines 7-21) makes clear what classes of events are of significance for establishing the required context. All event-emitting behaviour declarations are local to the aspect, as are the methods used to drive the recognition of context. However, AspectJ provides no improvement over the object-oriented approach in terms of how context is recognized. The complexity of the code for recognizing patterns of events limits one's ability to clearly understand the implementation and the intended concern.

The improved localization of event emission and customization also allows one to more easily change what events should be considered, and what behaviour should be invoked should context be established. However, while the aspect-oriented solution is localized, localization does not imply easy evolution. The AspectJ implementation is still difficult to evolve. Just as it was with the object-oriented solution, estab-

lishing context requires a low-level consideration of the relationships between events. Putting these relationships down in code is difficult, error-prone, and leads to an implementation that is difficult to understand and evolve.

In this section we considered aspect-orientation as a possible solution to the problems plaguing a purely object-oriented realization of our path-specific customization. What we show is that while aspect-oriented programming languages help to a certain degree, specifically by supporting localized non-invasive change, they do not modularize context-sensitive concerns completely. They lack a means to realize, in a way that retains our ability to locate, reason about, and change our context-sensitive concerns at the aspect level.

2.4 Summary

Three important characteristics of a program's implementation—traceability, comprehensibility, and evolvability—where demonstrated to be lacking in both object-oriented and aspect-oriented realizations of a simple context-sensitive concern. The object-oriented solutions exhibited scattering and tangling as well as complex and difficult to follow code for establishing and maintaining execution context. The aspect-oriented solution, while sufficiently addressing the scattering and tangling that plagued the object-oriented approaches, could not ease the difficult task of recognizing and maintaining context. The resulting implementation, while localized, remained hard to understand and change.

In conducting this demonstration, we have justified the thesis of our work, and motivated the need for better programming support for context-sensitive concerns.

Chapter 3

Declarative Event Patterns

Declarative event patterns (DEPs) are a language mechanism for modularizing context-sensitive concerns while retaining comprehensibility, traceability, and evolvability. They aim to make changing context-establishing code nearly as easy as changing a specification of context, encouraging exploration and easy evolution of context-sensitive concerns. In this chapter, we first provide a conceptual overview of our approach in Section 3.1. The DEPs approach itself is introduced in Section 3.2, where we discuss our constructs for specifying and implementing communicative context and present the syntax and informal semantics of these constructs concretely. We conclude this chapter with an application of DEPs to the path-specific customization of our Columba motivational example; this is done in Section 3.3.

3.1 Conceptual Overview

The idea of abstracting the dynamic execution of a program as a sequence of primitive events and allowing developers to specify context-establishing patterns of events to be matched against these sequences is at the heart of our approach. DEPs are language constructs that permit developers to specify, as code, the patterns of events necessary to establish context. Additionally, they allow developers to specify behaviours to occur when context has been established. And importantly, this is all done in a modular way that retains comprehensibility, traceability, and evolvability.

Primitive Events When establishing context, it is usually necessary to only consider a handful of events that mark points of interest in a program's execution. Conceptually, these contextual markers can be any observable event in the execution of a program. More practically they are often the entry or exit from method calls. Sometimes they may involve the entry or exit to the execution of a method's body or the assignment of a value to a field. Other times, primitive events may provide valued reference to the context surrounding the event. Generally though, the significance of an event is dependent on the context. They are often scattered amongst the different modules of a system.

Abstract Traces The sequence of events that declarative event patterns operate over are not a complete account of all events in the execution of a given program. Instead, these abstract traces are subjective and include only the execution events of interest to the pattern and exclude all others. By excluding non-relevant events from the trace, developers are able to focus on the essential nature of the pattern.

Expressivity The nature of context-establishing patterns in object-oriented systems is what we have looked to address with DEPs. As a result, the event patterns expressible by DEPs are equivalent to the languages expressible using context-free grammars. We aimed to be able to describe patterns involving sequential events, as well as nested event structures analogous to recursive procedure call structures should they be needed. It has been shown that context-free grammars are useful for modelling program execution [72, 85].

DEP Extensions to AspectJ When comparing the object-oriented solution to the Columba path-specific customization with the aspect-oriented one, it can be seen that some benefits were gained by leveraging AspectJ's advice mechanism to localize event instrumentation. Rather than start anew, DEPs extend the AspectJ language with new constructs specific for establishing and leveraging context. In particular, we

have added *tracecuts* (for establishing context), and the *history* primitive pointcut (for leveraging context).

Each tracecut is a declarative event pattern specification. *Primitive tracecuts* describe the individual events in a context-establishing specification. *Ordered tracecuts* combine other tracecuts to compose more elaborate declarative specifications of context. The history primitive pointcut is a conditional pointcut that may be advised like any other pointcut; advice making use of the `history` pointcut is only applied when the execution context specified by a target tracecut matches the program's actual execution.

3.2 Syntax and Semantics

Extending AspectJ, the constructs provided by our approach together aid in establishing and leveraging communicative context. In this section we take a concrete look at our extensions. Named tracecut declarations are introduced in Section 3.2.1. The syntax and informal semantics of primitive tracecuts are described in Section 3.2.2. A description of ordered tracecuts are found in Section 3.2.3. The `history` construct is covered in Section 3.2.4. Beyond simply establishing communicative context, DEPs provide features for exposing data surrounding an events occurrence, which we cover in Section 3.2.5, as well as refining communicative context with additional constraints, which we cover in Section 3.2.6.

3.2.1 Tracecut Declarations

Declarative event patterns describe either the individual events (by way of primitive tracecuts) or patterns of events (by way of ordered tracecuts) in the execution of a program. Although we dedicate complete sections to both primitive tracecuts (see Section 3.2.2) and ordered tracecuts (see Section 3.2.3), it is useful to first explain the basic form either type of tracecut may take. Named tracecut declarations follow

the syntax shown in Table 3.1.

Non-terminal		Production Rule
tracecut_declaration	⟶	tracecut_header **::=** tracecut **;**
tracecut_header	⟶	**tracecut IDENTIFIER (** formal_parameter_list_opt **)** semantic_decls_opt
semantic_decls_opt	⟶	ϵ
	⟶	**{:** formal_parameter_list_opt **:}**

Table 3.1. *Named tracecut declaration syntax*

Named tracecuts may be declared in the body of an aspect, just as named pointcuts are declared in AspectJ. They are marked by the keyword `tracecut`, and consist of an identifier (the name that will be associated with the tracecut declaration), optional formal and semantic parameters used to expose context from the recognized execution context (see Section 3.2.6.1), and either a single primitive event, or a composite pattern of events. Just how one specifies these patterns of events is revealed in the coming sections.

3.2.2 Primitive Tracecuts

Primitive tracecuts define the lexemes of declarative event patterns, capturing individual events in the execution trace. Two primitive tracecuts are provided by the tool. Entries into a join point can be captured through the use of the `entry` primitive tracecut, while exits from a join point can be captured through the use of the `exit` primitive tracecut. Both of these take a pointcut as an argument, which can expose state to the advice implementation through formal parameters, in the standard AspectJ fashion. The syntactic forms a primitive tracecut may take on are listed in Table 3.2.

In AspectJ, a pointcut may be specified in conjunction with after advice to capture the returning of a value from a method, or the throwing of an exception. Primitive

Non-terminal		Production Rule
primitive_tracecut	⟶	**entry** (pointcut_expr)
	⟶	**exit** (pointcut_expr)
	⟶	**exit** (pointcut_expr) **returning** (formal_parameter_opt)
	⟶	**exit** (pointcut_expr) **throwing** (formal_parameter_opt)

Table 3.2. *Primitive tracecut syntax demonstrating the* ***entry*** *and* ***exit*** *primitive tracecuts*

`exit` tracecuts allow one to capture these events in a way syntactically equivalent to AspectJ. Optionally, the actual returned value or thrown exception may be captured by specifying a formal parameter within the trailing parentheses.

3.2.2.1 Examples

The tracecut equivalents for those events used in the Columba example are listed in Figure 3.1. Recall that primitive tracecuts work to specify individual events in a program's execution, and are the combination of a regular AspectJ pointcut and an indication of entry or exit. This example contains a tracecut for each of the events corresponding to `observeInitHeaderEntry`, `observeInitHeaderExit`, `observeGetFromEntry`, and `observeGetSentFolder` in Chapter 2.

```
1 public aspect ColumbaCustomization{
2     // Declare join points of interest
3     pointcut initHeader(): call(void ReplyCommand+.initHeader(AbstractMessageFolder, Object[]));
4     pointcut getFrom(): call(Address[] BasicHeader.getFrom());
5     pointcut getFolder(): call(IFolder FolderCommandReference.getFolder(..));
6
7     // Context establishing events
8     tracecut initHeaderEntry()::= entry (initHeader());
9     tracecut initHeaderExit()::= exit (initHeader());
10    tracecut getFromEntry()::= entry (getFrom());
11    tracecut sentFolderExit()::= exit (getFolder()) returning (IFolder folder){:
12      if(folder.getUid() != 104) { fail; }
13    :}
14 }
```

Figure 3.1. *Tracecut equivalents to the context establishing events in the Columba example*

Within this DEP-augmented aspect, four named tracecuts have been declared. The first three of these (`initHeaderEntry`, `initHeaderExit`, and `getFromEntry`) demonstrate the use of the `entry` and `exit` primitive tracecuts. They make use of the pointcuts `initHeader` and `getFrom`. As before, the `initHeader` pointcut specifies method call join points corresponding to methods named `initHeader` on `ReplyCommand` or any of its subclasses. The `getFrom` pointcut similarly captures call join points of methods named `getFrom` on `BasicHeader`.

The `getFolder` pointcut, which is used in our last tracecut `sentFolderExit`, permits the selection of join points corresponding to calls to methods named `getFolder` on the `FolderCommandReference` type. Using the exit-returning tracecut form, `sentFolderExit` exposes the folder returned from a call to the `getFolder` method, and then conditionally accepts the event if the folder identifier is that of the **Sent** folder (104). In addition to demonstrating the exit-returning form, the `sentFolderExit` tracecut makes use of some advanced features of tracecuts, which we will come back to in Section 3.2.5.

In addition to the two primitive tracecuts described above, two special *anchoring-tracecuts* are provided. These so-called anchoring tracecuts provide a means to anchor more complex execution patterns into place. The caret (^) matches the beginning of the communication history; a dollar sign ($) matches the (current) end of the communication history.

3.2.3 Ordered Tracecuts

Ordered tracecuts provide a means to describe potentially complex execution patterns involving precedence and dominance of events. They are used to declare patterns on the temporal or causal ordering of events, and may be defined as potentially recursive patterns involving primitive tracecuts or references to named tracecuts. Concatenation, Kleene closure, disjunction, recursion, and various forms of syntactic sugar are provided. Kleene closure (*) is used to provide repetition of events. Square brackets

([and]) are used to specify optional tracecut parts. We use disjunction (|) to specify alternatives between tracecut clauses. DEP patterns are limited to those expressible by context-free languages. If one considers primitive tracecuts the terminal symbols of a context-free grammar, then ordered tracecuts can be seen as the production rules of a context-free grammar. The syntactic forms an ordered tracecut declaration may take on adhere to the partial grammar shown in Table 3.3. The various facets of this grammar are revealed throughout this chapter.

Non-terminal		Production Rule
tracecut	⟶	disjunctive_tracecut
disjunctive_tracecut	⟶	disjunctive_tracecut | disjunct
	⟶	disjunct
disjunct	⟶	event_list
	⟶	event_list semantic_block
	⟶	semantic_block
	⟶	ϵ
event_list	⟶	kleene_closure
	⟶	event_list kleene_closure
kleene_closure	⟶	kleene_plus $\star$
	⟶	kleene_plus
kleene_plus	⟶	event +
	⟶	event
tracecut_reference	⟶	**IDENTIFIER** (type_id_star_list_opt)
nested_tracecut	⟶	[tracecut]
	⟶	(tracecut)
event	⟶	$
	⟶	^
	⟶	primitive_tracecut
	⟶	nested_tracecut
	⟶	tracecut_reference
semantic_block	⟶	**{:** block_statements_opt **:}**

Table 3.3. *Tracecut syntax*

3.2.3.1 Examples

On the surface, ordered tracecuts resemble the AspectJ pointcut. Unlike a regular pointcut, that specifies the set of valid join points of interest, a tracecut specifies the set of valid *patterns of events*. Ordered tracecuts can be composed from other ordered and primitive tracecuts. Tracecuts may be named and referred to by these names for improved clarity and comprehensibility. The sequence of events that declarative event patterns operate over are not a complete account of all events in the execution of a given program. They are abstract traces that include only the primitive events of interest to the tracecut and exclude all others.

In Figure 3.2, a DEP-augmented aspect is declared that demonstrates a complex ordered tracecut in use. Here four primitive entry pointcuts are declared, one for each entry to the call of methods `aMethod`, `bMethod`, `cMethod`, and `dMethod`. The tracecut named `complex` declares an ordered tracecut that specifies that execution paths initiated at the beginning of execution (via the anchoring caret symbol), up to the current point in execution (via the anchoring dollar symbol), in which first an `a` event occurs, followed by zero or more `b` events, optionally followed by another `a` event, and then either a `c` event or a `d` event. This example demonstrates the use of some of the syntactic sugar provided in the DEP language extensions. In Figure 3.3 we present an example of an ordered tracecut which uses recursion.

```
public aspect ComplexOrderedTracecut {
   tracecut a() ::= entry( call (void aMethod(..))) ;
   tracecut b() ::= entry( call (void bMethod(..))) ;
   tracecut c() ::= entry( call (void cMethod(..))) ;
   tracecut d() ::= entry( call (void dMethod(..))) ;
   tracecut complex() ::= ^ a() b()* [a()] ( c() | d() ) $ ;
```

Figure 3.2. *A DEP-augmented aspect containing a complex ordered tracecut* `complex`*, composed of four primitive tracecuts.*

In Figure 3.3, a DEP-augmented aspect contains a tracecut named `nesting` that

describes a pattern of events recursively. The pattern `nesting` describes a self-embedding between the primitive events `a` and `b`. `nesting` may alternatively (via disjunction) simply match against the empty stream of events (the last disjunctive part is empty). Some possible steams of events that this pattern would match include the following: (ϵ), $(a \longrightarrow b)$, $(a \longrightarrow b)$, $(a \longrightarrow a \longrightarrow b \longrightarrow b)$ and so on.[1]

```
public aspect ComplexOrderedTracecut {
  tracecut a() ::= entry( call (void aMethod(..))) ;
  tracecut b() ::= entry( call (void bMethod(..))) ;
  tracecut nesting() ::=  a() nesting() b()
                      |   ;
```

Figure 3.3. *A DEP-augmented aspect containing a complex ordered tracecut* ***`nesting`****, which recursively defines itself.*

3.2.4 History Primitive Pointcut

In order to leverage the context specified by tracecuts, the `history` primitive pointcut was added. This pointcut, like regular AspectJ pointcuts, is used to specify when advice should be applied in the execution of a program. It constrains the application of advice to those times when the context of a target tracecut holds. In this way, advice application may be refined to leverage execution contexts. Each `history` pointcut has a target tracecut; each target tracecut is recognized independently. Targets can be references to named tracecuts, or anonymous tracecut specifications. The syntax for the `history` pointcut is shown in Table 3.4.

Non-terminal		Production Rule
basic_pointcut_expr	⟶	**history** (tracecut)

Table 3.4. *History pointcut syntax*

[1] We use the ⟶ symbol to denote zero or more communication events in the execution that are not a part of the events that make up the DEP pattern.

In Figure 3.4 we apply the `history` pointcut to advise a set of method call join points, but only when a certain execution situation has occurred. In this example before advice is applied to methods that are named beginning with `get` defined on any class, with any number of parameters, and return type `void`, but only if the `nesting` pattern (as defined in Figure 3.3) has been previously observed.

```
HistoryExample {
 pointcut methodCalls() : call(void *.get*(..));
 before(): history(nesting()) && methodCalls() {
   System.out.println("The nesting pattern was observed.");
 }
```

Figure 3.4. *A brief example of the usage of the history primitive pointcut. Before advice is applied to methods that are named beginning with* ***`get`*** *defined on any class, with any number of parameters, and return type* ***`void`****, but only if the* ***`nesting`*** *pattern has been previously observed.*

Conceptually a straight-forward pointcut extension, the `history` designator allows one to take advantage of DEPs in regular AspectJ advice. As a result, the DEP developer is able to refine advice application to specific execution contexts in a way not possible with standard AspectJ.

3.2.5 Context Exposure

Beyond acknowledging that a given communicative context holds, we often want to establish references to objects and values by their historic placement in context.

One of the most powerful features of AspectJ is the ability to expose the state that exists at a join point for use within advice; this is known as context exposure (see Section 2.3.1.3). Context exposure is also available with both primitive and ordered tracecuts. In Figure 3.5, we define a primitive tracecut that exposes state from a pointcut argument. The pointcut in this case is named `simplePc` and the primitive tracecut that uses this pointcut as argument and source of context is named

`simpleEntry`. Here the `simpleEntry` tracecut forwards exposed context from the `simplePc` argument. The tracecut `simpleEntry` forwards the exposed integer i of pointcut `simplePc` through its formal parameter `j`.

```
simplePc(int i): call(* *.simple(int)) && args(i);
simpleEntry(int j) ::= entry(simplePc(j));
```

Figure 3.5. *In this code snippet, a primitive tracecut forwards exposed context from its pointcut arguement. The tracecut* ***simpleEntry*** *forwards the exposed integer i of pointcut* ***simplePc*** *through its formal parameter* ***j****.*

Context exposure for ordered tracecuts works in a similar fashion. An ordered tracecut such as `exposingTracecut` in Figure 3.6 can forward the exposed state from its composite tracecuts and even to a piece of advice through the use of a `history` pointcut. This ordered tracecut exposes two integers bound to those exposed by the two constituent primitive tracecuts. The advice declared here uses the `history` pointcut to expose these values to the advice body.

```
 exposingTracecut(int i,int j)::= simpleEntry(i)  exit(simplePc(j));
(int a, int b) : history(orderedTracecut(a,b))
                    && execution(void Example.someMethod()){
  // Do something with the exposed context
  System.out.println("a+b=" + (a+b));
```

Figure 3.6. *This ordered tracecut exposes two integers bound to those exposed by the two constituent primitive tracecut parts. The advice declared here uses the* ***history*** *pointcut to expose these values to the advice body.*

3.2.6 Constraining and Altering Context

Sometimes we may wish to constrain the communicative context, not solely by which pattern of communication has occurred, but also on the significance of the state surrounding events, or based on some arbitrary computation on that state. We have built into DEPs mechanisms for achieving this flexibility. These include *semantic code*

declarations (Section 3.2.6.1), the ***fail*** *statement* (Section 3.2.6.2), and *free variables* (Section 3.2.6.3).

3.2.6.1 Semantic Code Declarations

As a matter of convenience, in addition to capturing and exposing state solely through formal parameters, DEPs provide a way to declare temporary semantic variables. Like a tracecut's formal parameter, state exposed can also be bound to these variables. Both formally bound variables and temporary semantic variables can be manipulated in an optional *semantic block* of Java code. These features in combination may be used to alter what state is bound to a tracecut's formal parameters, or provide any arbitrary computation to take place at the point in time that the associated tracecut pattern is recognized.

For primitive tracecuts, the use of temporaries and the semantic action block typically involves modifying the state drawn from the join point before assigning it to a formal parameter. Temporary local variables are declared in a block on the left-hand-side of a named tracecut declaration as seen in Figure 3.7. In this example, a primitive tracecut `primitiveTracecut` exposes an `Integer` argument passed to a call to method `a`, but before exposing it, it ensures that the argument is not `null`. If the exposed argument is `null`, a new `Integer` object is created and bound to formal parameter `i` of tracecut `primitiveTracecut`.

Notice that in addition to declaring a formal parameter `Integer i`, the tracecut `primitiveTracecut` declares a temporary variable `Integer exposed`, enclosed in {: :} pairs. Similarly, the semantic code block is distinguished by this parenthetic form.

For ordered tracecuts, temporaries may be used to hold values exposed by the parts of the tracecut, and are used in much the same way as in the primitive tracecut case. In Figure 3.8 an ordered tracecut `helloWorld` exposes the concatenation of two strings exposed by its tracecut parts. The tracecuts `hello` and `world` expose strings at potentially dispersed points in the execution. This tracecut brings these exposed

```
 pc(Integer arg): call(* *.a(Integer)) && args(arg);
 primitiveTracecut(Integer i) {: Integer exposed :}
=entry(pc(exposed))
 {:
    if(exposed == null){
      i = new Integer(0);
    } else {
      i = exposed;
    }
 :};
```

Figure 3.7. *A primitive tracecut* ***primitiveTracecut*** *exposes an* ***Integer*** *argument passed to a call to method* ***a****, but before exposing it, it ensures that the value of the argument is not* ***null****. If the exposed argument is* ***null****, a new* ***Integer*** *object is created and bound to formal parameter* ***i*** *of tracecut* ***primitiveTracecut****.*

values together for further computation.

```
helloWorld(String hw) {: String h, String w:} ::= hello(h) world(w)
{:
   hw = h + w;
:};
```

Figure 3.8. *An ordered tracecut* ***helloWorld*** *exposes the concatenation of two strings exposed by its tracecut parts. The tracecuts* ***hello*** *and* ***world*** *expose Strings at potentially dispersed points in the execution. This tracecut brings these exposed values together for further computation.*

Semantic action blocks permit actions to be taken before a pattern is completely matched. At the point where the sub-pattern is recognized, the action block is executed.

3.2.6.2 Failure

In addition to providing a means to alter exposed context, or perform arbitrary computations at the time a pattern is matched, one can also reject the occurrence of a particular pattern along the current execution path, resulting in a discontinued consideration of that path. By using the special `fail` statement from within a semantic

action block one can conditionally reject the recognition of a tracecut.

We can enforce a semantic constraint on the class of events selected by the primitive tracecuts. For example, in Figure 3.9, `i` must be an even number. Rejection is indicated by the use of the identifier `fail`, which also causes execution of the semantic block to end. An explicit `return` or falling off the end of the semantic block (an implied `return`) indicate no semantic failure for a match.

```
simplePc(Integer arg): call(* *.simple(Integer)) && args(arg);
simpleEntry(int i)
{: Integer arg :} ::= entry(simplePc(arg))
{:
   i = arg.intValue();
   if(i & 1) // bit operation
     fail; // reject this occurrence
:};
```

Figure 3.9. *The entry primitive tracecut exposes the* ***`Integer`*** *argument passed to calls of simple; failure is used to reject the occurrence of this event.*

The use of failure in semantic blocks associated with ordered tracecuts indicates that the current avenue of recognition should be abandoned. The semantic action block, as it was with the primitive tracecut, ends if `fail` is reached. Consider Figure 3.10; here an ordered tracecut `requestReply` imposes a semantic constraint on the message identifier of a `Request`, and of a `Reply`. If the message identifier of the `Reply` does not match that of the `Request`, `requestReply` will fail to match.

```
requestReply()
{: Request req, Reply reply :} ::=
requestTracecut(req) replyTracecut(reply)
{:
   if(req.msg_id != reply.msg_id) fail;
:};
```

Figure 3.10. *An ordered tracecut* ***`requestReply`*** *imposes a semantic constraint on the message identifier of a* ***`Request`****, and of a* ***`Reply`****. If the message identifier of the* ***`Reply`*** *does not match that of the* ***`Request`****,* ***`requestReply`*** *will fail to match.*

3.2.6.3 Free Variables

To DEPs we have also added *free variables*. Formal or temporary variables may be used to specify binding constraints on exposed state. If a variable appears more than once in a tracecut as a binding to exposed context, the first exposure is bound to the variable, and subsequent exposures are implicitly accepted or rejected based on the value of the exposed context.

```
acecut a(Relation x)::= entry(call(void Relation.a_method() && target(x)));
acecut b(Relation y)::= entry(call(void Relation.b_method() && target(y)));
acecut c(Relation z)::= entry(call(void Relation.c_method() && target(z)));

 initialisationSequence()
 {: Relation instance :} ::= a(instance) b(instance) c(instance);
```

Figure 3.11. *An ordered tracecut makes use of free variables to ensure that a sequence of call events are targetted to the same instance.*

In Figure 3.11 we illustrate the use of free variables. Here we have defined three primitive tracecuts `a`, `b`, and `c`. These tracecuts correspond to entry events on calls to methods `a_method`, `b_method`, and `c_method` all of which are defined on type `Relation`. Each tracecut additionally exposes the target instance of the call, and binds it to a formal parameter. A fourth tracecut, `initialisationSequence`, specifies an ordering of these events in order to observe the sequence `a b c`. By declaring the temporary semantic variable `instance`, and using it as the binding target for each of the three context exposing events specified, the pattern is restricted to only `a`, `b`, and `c` events which target `instance`. When event `a` occurs, `instance`, currently unbound, is then bound to the `Relation` instance exposed through `a`. Now when event `b` occurs, its exposed context is checked against the currently bound `instance`. If it is equivalent, event `b` is accepted and observed, otherwise event `b` is not observed. The same goes for event `c`. Only if the exposed state is equal to that of the bound value, will it be observed.

Using free variables allows for the expression of communicative context not solely

on the types of events and their ordered occurrence, but also on consistent instantiating of variables bound to exposed context. In some situations however, we may wish to permit a `rebinding` of a variable, particular in the cases when a variable is used as the binding target in a repetitive pattern. Take for example Figure 3.12. Here we have two basic tracecut patterns involving the exposure of `String s`—repetitively.

```
acecut atom(String string)::= entry(call(void foo.bar(String) &&
                                 args(string));
acecut pattern1(String s)::= atom(s)*;
acecut pattern2(String s)::= atom(rebind(s))*;
```

Figure 3.12. *Rebinding variables in DEPs*

The first pattern (`pattern1`) involves the repetitive occurrence of zero or more `atom` events exposing `s`. Further to this, each `s` *must* be the same. The second pattern (`pattern2`) on the otherhand permits `s` to be *rebound* on each occurrance of `atom`, allowing `s` to be the most recently exposed `String` argument of `atom`. `rebind` acts to bypass normal variable binding constraints, permitting for a new binding to occur. It has no effect on unbound variables.

3.3 DEP solution to Columba customization

With the constructs provided by DEPs fully explained, we are now in a position to demonstrate how we might apply them to the problem of path-specific customization in our Columba case study. Recall from Chapter 2 that neither object-oriented nor aspect-oriented approaches managed to fully modularize our path-specific customization.

Figure 3.13 shows the complete path-specific customization using DEPs. Like the AspectJ solution, this one is contained in a single aspect. The base program was not altered in any fashion. Lines 1-14 are identical to those explained in Figure 3.1, and can be summarized by saying that they declare those events of significance to

```
1 public aspect ColumbaCustomization {
2
3     // Declare join points of interest
4     pointcut initHeader(): call(void ReplyCommand+.initHeader(AbstractMessageFolder, Object[]));
5     pointcut getFrom(): call(Address[] BasicHeader.getFrom());
6     pointcut getFolder(): call(IFolder FolderCommandReference.getFolder(..));
7
8     // Context establishing events
9     tracecut initHeaderEntry()::= entry (initHeader());
10    tracecut initHeaderExit()::= exit (initHeader());
11    tracecut getFromEntry()::= entry (getFrom());
12    tracecut sentFolderExit()::= exit (getFolder()) returning (IFolder folder){:
13      if(folder.getUid() != 104) { fail; }
14    :}
15
16    // Specification of contextual pattern
17    tracecut customizationContext()::=
18      [initHeaderExit()] sentFolderExit() initHeaderEntry() getFromEntry()+ $;
19
20    // Leveraging of context to apply customization
21    Address[] around(BasicHeader header):
22      execution(Address[] BasicHeader.getFrom())
23      && this(header)
24      && history(customizationContext()){
25        return header.getTo();
26    }
27 }
```

Figure 3.13. *Tracecut solution to the full path-specific customization in the Columba example*

the contextual pattern. Now that we have introduced context exposure and tracecut failure, lines 12-14 can be more thoroughly explained. The tracecut declared here captures exit events returning a result of type `IFolder`, which is to be bound to the variable `folder`. The value of `folder` is compared against the **Sent** folder constant (`104`). If it is not the **Sent** folder, `fail` is called, and the primitive tracecut is rejected. On line 24, we leverage the context specified by `customizationContext`; it is used as a target tracecut to a `history` pointcut. The usage refines the application of our customization advice to just those contexts allowed by our tracecut.

We ultimately want to apply our customization only when the execution context conforms to the following pattern.

$$(\texttt{initHeader}^{call}_{exit})?\ (\texttt{getFolder}^{call}_{exit})\ (\texttt{initHeader}^{call}_{entry})\ (\texttt{getFrom}^{call}_{entry})^{+}\ \$$$

The traceability from our regular expression specification to its realization in code is evident in lines 17-18. Here we declare an ordered tracecut named `customization-Context`, that appropriately enough, specifies the communicative context required for establishing our Columba path customization. This tracecut is repeated below to emphasis how it resembles our specification directly.

$$[initHeaderExit()]\ sentFolderExit()\ initHeaderEntry()\ getFromEntry()+\ \$$$

It is easy to compare our tracecut declaration with the original specification. Because of this resemblance, reasoning about the tracecut is nearly the same as reasoning about the specification itself. If a change to the specification is required, changing the tracecut should involve effort proportional to that change. In Chapter 5 we conclude the Columba study with a full discussion of this solution.

3.4 Summary

In this chapter we introduced the goals, syntax and informal semantics of the language constructs that make up the declarative event pattern extension to AspectJ.

The idea of abstracting the dynamic execution of a program as a sequence of primitive events and allowing developers to specify context-establishing patterns of events to be matched against these sequences is at the heart of our approach. DEPs are language constructs that permit developers to specify, as code, the patterns of events necessary to establish context. Additionally, they allow developers to specify behaviours to occur when context has been established. Primitive tracecuts represent basic communication events. They can be combined to form ordered tracecuts, which provide a declarative means of realizing communicative context. Using the `history` pointcut, normal AspectJ advice can determine if a execution context described by a tracecut holds. Acceptable execution traces may be semantically rejected or further constrained with the use of semantic actions, fail, or free variables. Additionally, the `history` pointcut can be used to draw state values from past executions forward for exposure in advise. The benefits gained by our DEP constructs when applied to the Columba customization problem presented in Chapter 2 are shown. The correspondence between our specification of context, and our declaration of context using DEPs is self-evident.

Chapter 4

An Implementation of Declarative Event Patterns

In order to experiment with the ideas of DEPs, we have constructed a proof-of-concept tool called URD [1]. From a practical perspective, URD operates in much the same way as a parser generator (such as YACC [46] or Bison [23]). It translates declarative specifications of event patterns (tracecuts), expressed as context-free languages, into (1) event parsers for recognizing such patterns, (2) the instrumentation code that will announce the occurrence of particular events at run-time to these event parsers, and (3) the specification of the points in the source code where the instrumentation must be injected.

We begin by discussing the high-level structure of the URD tool in Section 4.1. URD performs preliminary analyses and transformations; the most prominent of these are covered in Section 4.2 and Section 4.3. The details regarding how the URD-generated event parsers are constructed and used to establish context are explained in Section 4.4.

[1] Originally, URD was an acronym, but its derivation has been lost. However, the name does coincidentally correspond nicely to a figure in Norse mythology. As once source puts it, "Urd is the first of three sisters known as the Norns (Urd, Skuld, and Verdandi); they are roughly equated to the Greek fates (Past, Present, Future) [57]." The word Urd is apparently synonymous with "Wyrd", an Anglo-Saxon word for the unseen influences behind events [95].

4.1 High-Level Structure

URD is a source-to-source transformation tool that takes AspectJ 1.2 compliant aspects augmented with our DEP constructs and transforms them into standard AspectJ source code. The resulting source code contains all the necessary details for recognizing individual events as they occur, as well as for recognizing context establishing patterns over these events.

The main controller `urd.compiler` is the entry point to URD. It is a command-line tool that processes input consisting of any number of DEP-augmented aspect source files with the extension `.urd`. These `urd` files are processed individually; each file is parsed and an abstract syntax tree (AST) representation is constructed. This AST undergoes preliminary transformations and is analysed in order to locate and extract tracecut information; standard AspectJ declarations are preserved. For each tracecut that is the target of a `history` pointcut, URD generates the appropriate event parser code declarations and recombines these declarations with the AST. Finally the AST is further transformed to output a valid standard AspectJ source file corresponding to each `urd` input file. The resulting standard AspectJ code relies on the presence of an accompanying runtime that provides the supporting infrastructure necessary to drive our generalized parsing algorithm.

4.2 Analysis and Early Transformations

URD initially constructs an abstract syntax tree representation (AST) for each of the input files in turn. The AST undergoes a number of minor transformations in order to prepare for the extraction of a proper context-free grammar for each of the tracecut declarations that serve as targets to `history` pointcuts. These minor transformations are summarized below.

Nomination of nameless terminals Anonymous primitive tracecuts (unnamed occurrences of `entry` & `exit` primitives) are given names. This pass generates declarations for anonymous primitive tracecuts and replaces these anonymous occurrences with tracecut references to the new name. If the anonymous tracecut exposes context to bind to variables, care is taken to properly bind this exposed context in the new explicit declarations. As an example, consider the following tracecut that contains the anonymous occurrence of two primitive tracecuts.

```
 namelessTerminals() ::=
exit(call(void foo.bar())) exit(call(void foo.baz()));
```

The above tracecut would be transformed into the three tracecuts below:

```
acecut urd$TERMINAL$0() ::= exit(call(void (foo).bar()));
acecut urd$TERMINAL$1() ::= exit(call(void (foo).baz()));
acecut namelessTerminals() ::= urd$TERMINAL$0() urd$TERMINAL$1();
```

The anonymous primitive tracecuts have been replaced with references to newly declared tracecuts `urd$TERMINAL$0` and `urd$TERMINAL$1`.

Nomination of nameless non-terminals Like the previous pass, this one names ordered tracecuts that are nested within other tracecuts. Again, care is taken to make certain that exposed state is properly carried through in the new declarations. The following tracecut exhibits a nested ordered tracecut; it makes use of the unmentioned tracecuts `a`, `b`, and `c`.

```
acecut namelessNonTerminals() ::= ( a() b() | ) | c();
```

The result of this transformation would be as follows.

```
acecut namelessNonTerminals() ::= urd$anon$0() | c();
acecut urd$anon$0() ::=  a() b() | ;
```

The nested anonymous tracecut is replaced with a reference to the equivalent named tracecut `urd$anon$0`.

A similar transformation occurs for occurrences of semantic action blocks. In the example below, an inline semantic action is presented along with its transformation.

```
acecut inlineActionBlock() ::=  {: println("hello"); :} b() | c();
```

The result of this transformation would be as follows. As before, variable bindings are considered when performing these transformations.

```
acecut inlineActionBlock() ::= urd$anon$0() b() | c();
acecut urd$anon$0() ::=  {: println("hello"); :};
```

In this case, the semantic action block has been transformed into a empty rule with associated action.

Expansion of syntactic sugar Syntactic sugar, such as Kleene closure, Kleene plus, and optional parts are expanded in this pass. New ordered tracecut declarations are added to represent each of these forms.

```
acecut syntacticSugar() ::= a()+  | b()* | [c()];
```

The above tracecut transformed would be as follows.

```
acecut syntacticSugar() ::= urd$anon$1() | urd$anon$2() | c() | ;
acecut urd$anon$1() ::= a() | urd$anon$1() a();
acecut urd$anon$2() ::= b() | urd$anon$2() b();
```

New tracecuts are declared to specify the repetitions resulting from `a()+` and `b()*`. We use a left-recursive pattern for repetition for reasons exampled in Section 4.4. The optional `[c()]` yields a clause where `c()` is required, as well as an empty clause where it is not. This same empty clause accounts for the case when `b()*` would match the empty string of events.

Grammar extraction With the tracecuts properly named and expanded, this phase extracts a context free grammar for each target tracecut. A transitive closure is performed; the reachable tracecuts (ordered and primitive) form the rules and lexical elements of a context free grammar. This information is maintained for future use in constructing the context establishing parser.

Semantic Actions, exposure and constraints For each production rule a further pass is conducted to collect information regarding what actions should be executed if and when the rule is recognized, what context (if any) the rule exposes, and what binding constraints (if any) should be applied. This information is stored for later use in the generation of the final resulting aspect. Ordered tracecut declarations are pruned from the AST upon completion of this phase.

4.3 Transformations

URD performs an number of transformations on the AST representation before it resembles a valid AspectJ representation. These transformations revolve around recombining the aspect with the generated parsers. They provide events to the parsers, expose state for use by the parsers, and consult the parsers to leverage context. URD automatically transforms primitive tracecut declarations into standard AspectJ advice. We demonstrate the variations of transformation in Section 4.3.1. Leveraging context through the use of the `history` pointcut is an important facet of our DEPs approach. In Section 4.3.2 we look at how the code is altered to support this. Semantic action code associated with ordered tracecuts is written within the context of the enclosing aspect. In Section 4.3.3 we outline how this code is transformed while correctly maintaining the *self* reference.

4.3.1 Primitive Tracecut transformations

The primitive tracecut transformation process provides instrumentation to announce the occurrences of primitive events to the event parser. In addition to providing a stream of events to the parser, references to exposed context at primitive tracecut join point locations must be maintained. Semantic action code must also be given a chance to execute, and optionally `fail` (see Chapter 3, Section 3.2.6.2). In this section we look at how various facets of primitive tracecuts are transformed.

The Basic Transformation The basic primitive tracecut transformation is as follows. Recall that AspectJ provides a means to augment existing code to provide different or additional behaviour. With URD, all `exit` tracecuts correspond to `after` advice on the pointcut specified by the tracecut. Similarly, `entry` tracecuts correspond to `before` advice.

```
acecut basic()::= exit(call(void (*).user(String)));
```

The `exit` primitive tracecut above would be transformed into the event-emitting `after` advice seen below.

```
ter():
  call(void (*).user(String)) {
      urd$goal$3$parser.nextToken(new
              urd.runtime.parser.UrdToken(urd$goal$3_id_basic));
```

Note that two fragments of the transformed code have been emphasized. The first box is the name of a generated parser instance; in this case the parser `urd$goal$3$parser` is interested in primitive event types indicated by a generated constant identifier `urd$goal$3_id_basic`, the second highlighted fragment.

The Context Exposing Transformation Locally exposed context is added to a *binding environment*—a shared stack structure for maintaining bound context variables—and passed along with an indication of event type to the parser.

```
acecut exposed(String s)::= exit(call(void (*).user(String))
 && args(s));
```

The context exposing primitive tracecut above would be transformed into the event-emitting `after` advice seen below.

```
ter(String s):
  call(void (*).user(String)) && args(s) {
    urd.runtime.context.LocalBindings urd$local_bindings =
      new urd.runtime.context.LocalBindings(1);
    urd$local_bindings.addBinding(s);
    urd$goal$3$parser.nextToken(
      new urd.runtime.parser.UrdToken(urd$goal$3_id_exposed,
      urd$local_bindings));
```

Note the inclusion of the environment declaration `urd$local_bindings`. The exposed string `s` is added to the environment, and then the environment is passed as an additional parameter to the `nextToken` message to our parser instance.

The Semantic and Constrained Transformation If semantic actions are present they are executed before this information is handed off to the parser. This permits the manipulation of exposed context before it is bound to any variable and for the advice to be cut short if an explicit `fail` should occur.

```
acecut semanticAction(String s)::= exit(call(void (*).user(String))
&& args(s)){:
  if(!s.equals("okay")){ fail; }
;
```

The primitive tracecut above exposes a string `s`, and conditionally accepts or rejects the occurrence of this event on its value; if `s` is equal to `"okay"` then the event (and the exposed value of `s`) will be passed on to the parser. Otherwise the event will be ignored. The transformation is shown below.

```
after(String s):
  call(void (*).user(String)) &&
    args(s) {
      if (!s.equals("okay")) { return; }
      urd.runtime.context.LocalBindings urd$local_bindings =
        new urd.runtime.context.LocalBindings(1);
      urd$local_bindings.addBinding(s);
      urd$goal$3$parser.nextToken(new
      urd.runtime.parser.UrdToken(urd$goal$3_id_semanticAction,
      urd$local_bindings));
```

Note the nested boxes in this code fragment. The outer box contains the semantic action, and is positioned such that it will execute *before* variables are bound and the event is passed to the parser. If the semantic action code alters the bindings in some way, this position will allow the changes to take effect. The inner box highlights the transformation from `fail` to `return`. The explicit `return` from advice permits us to cut short the execution of the advice and avoid the binding and passing of information to the parser entirely—effectively ignoring the event.

Temporary semantic variables, when used in the semantic action code of primitive tracecuts, are integrated into the signature of the advice and bound to the exposed argument just as a normal formal parameter. In order to demonstrate this transformation, we present the tracecut below.

```
acecut semvar(String formal) {: String exposed :} ::=
exit(call(void (*).user(String))
  && args(exposed)){:
 if(exposed.length() > 0){
```

```
formal = exposed;
 } else {
formal = "invalid string";
 }
;
```

The modification incorporates a semantic variable `exposed`, and checks that the length of this string is greater than 0 before binding it to the formal parameter `formal`. Below is the transformed version.

```
ter(String exposed):
  call(void (*).user(String)) &&
    args(exposed) {
      String formal = null;
      if (exposed.length() > 0) {
        formal = exposed;
      } else {
        formal = "invalid string";
      }
      urd.runtime.context.LocalBindings urd$local_bindings =
        new urd.runtime.context.LocalBindings(1);
      urd$local_bindings.addBinding(formal);
      urd$goal$3$parser.nextToken(new
      urd.runtime.parser.UrdToken(urd$goal$3_id_semvar,
   urd$local_bindings));
```

With this transformation one will notice that the semantic variable `exposed` now appears in the advice signature. Also note that, because it appears unbound in the tracecut's pointcut, the formal parameter `formal` has been relegated to a local variable in the body of the advice[2].

This section has presented the variety of primitive tracecut transformations performed by the URD tool. These transformations play the role of generating event-emitting advice. In all cases, this advice communicates the occurrence of an event to

[2]AspectJ does not permit advice parameters to remain unbound.

all interested parsers, and optionally may pass exposed context to the parsers when it is required.

4.3.2 The History Transformations

The `history` pointcut conditionally indicates the acceptance of a tracecut to AspectJ advice. There are two forms of transformation of the `history` pointcut into standard AspectJ—one that exposes context, and one that does not.

The Basic History Transformation When a tracecut that does not expose context is used as the target of a `history` pointcut all that is required is a conditional check against the parser in order to determine if context has been established.

Below is a piece of advice that is intended for execute after a call to the `foo.bar` method, but only when the first occurrence of the primitive tracecut `basic` (introduced in the previous section on event-emitting transformations) has occurred. We have used an anonymous ordered tracecut (`^ basic()`); this will be given a name as part of the early transformation process.

```
ter(): call(void foo.bar()) && history(^ basic() ){
ystem.out.println("Observed first basic event");
```

The above advice results in the following transformed code.

```
ter(): call(void foo.bar()) &&
(aspectOf().urd$goal$0$parser.isAccept()) {
ystem.out.println("Observed first basic event");
```

There are two points of significance with this transformed advice. The first is the use of `aspectOf()`. The `if` pointcut may only access static references. The static `aspectOf()` method satisfies this requirement by providing a reference to the

aspect instance currently associated with the advised join point. From this reference, we obtain the parser of interest and call its `isAccept()` method. If the context is currently satisfied, this method returns true; otherwise, it returns false.

The Context-Exposing History Transformation When a tracecut exposes context via the target tracecut, additional transformations are required in order to bind the exposed state to the advice's formal variables. Sometimes URD may need to alter the body of the advice so as to declare and assign values to local variables to hold the exposed context. This is done in order to simulate advice parameters to work around AspectJ's strict assurance that formal parameters be bound in a pointcut declaration. Altering the above history application to expose a string `s` through an `exposed` event (introduced in the event-emitting section) leads to the following code.

```
ter(String s): call(void foo.bar()) && history(^ exposed(s) ){
ystem.out.println("Observed first exposed event");
```

The above advice results in the following transformed code.

```
fter():
  call(void (foo).bar()) &&
    if(aspectOf().urd$goal$1$parser.isAccept()) {
      try {
          String s =
            (String)
              urd$goal$1$parser.getAcceptBindings().getBindingAsObject(0);
          System.out.println("Observed first exposed event");
      }
      catch (urd.runtime.UrdRuntimeTypeCastException rttc) {  }
  }
```

As before, the advice is conditional on the state of the parser. But what is different here is that the formal string `s` is pushed into the advice body. The variable `s` is then

assigned a value drawn from the exposed values determined by the parser. It should be noted that due to AspectJ's enforcement that formal parameters be bound, the use of `proceed` within `around` advice using the `history` pointcut requires special attention. Formal parameters exposed via `history` pointcuts should not be taken into account when considering the argument arity of `proceed`.

This section has presented the two transformations carried out by URD in order to leverage context using the `history` pointcut. Each of these transformations consult the event parser corresponding to their target tracecut in order to determine if the requisite context has been established. If context is exposed through the `history` pointcut, references to this context must additionally be obtained.

4.3.3 Transformations for Ordered Tracecuts

When an ordered tracecut exposes state, the semantic actions that are associated with the tracecut as well as code for maintaining bound context must be given a chance to execute. Because each tracecut declaration is written in the context of the aspect, usage of the self-reference `this` from within a semantic action block, whether implicit or explicit, needs to resolve correctly. Therefore, for each clause in a grammar extracted for a target tracecut, we group semantic action code and the default context-binding mechanisms into a corresponding method on the originating aspect. These methods are called by the parser whenever the clause has been recognized. This section looks at these generated methods in their various forms.

The Basic Ordered Tracecut Transformation A tracecut that does not expose context and does not perform any additional semantic actions is presented below. Assume the presence of primitive tracecuts `a`, `b`, and `c`.

```
acecut basicOrderedTracecut()::= a() b() c();
```

The resulting method for this single clause tracecut provides no apparent utility. However, it is necessary for our parsing algorithm.

```
ivate urd.runtime.context.IEnvironment
basicOrderedTracecut$R0(urd.runtime.context.IEnvironment bindings)
   throws urd.runtime.UrdRuntimeException {
      try {
          if (true) { {  } }
          return new urd.runtime.context.LocalBindings(0);
      }
      catch (Exception e) {
          throw new urd.runtime.UrdRuntimeException(e);
      }
  }
```

Notice that it both takes as argument and returns an `IEnvironment` reference. These methods are responsible for maintaining (or overriding through the use of semantic action blocks) the exposed context. Since our example does not expose context it simply ignores the incoming environment and returns an empty environment in turn. The third boxed area is a placeholder for user defined semantic actions. We should also point out that one such method is generated for each *clause* of a tracecut. That is, each alternative rule in use by a tracecut.

The Context-Exposing Ordered Tracecut Transformation A tracecut that does expose context but does not perform any additional semantic actions is presented below. Assume the presence of primitive tracecuts `a`, `b`, and `c`. This time additionally assume that `a`, `b`, and `c` each expose a string value.

```
acecut exposingOrderedTracecut(String x, String z) {: String y :}
:=  a(x) b(y) c(z);
```

The above tracecut exposes two string values `x` and `z` and binds to an additional string through a semantic variable `y`. The first of these (`x`) is bound to the exposed

string from primitive tracecut a. The second string exposed by this ordered tracecut (z) is the one exposed by tracecut c. Variable y binds to the string exposed through b. The method generated to maintain this context is now presented.

```
ivate urd.runtime.context.IEnvironment
exposingOrderedTracecut$R0(urd.runtime.context.IEnvironment bindings)
   throws urd.runtime.UrdRuntimeException {
      try {
          String x = (String) bindings.getBindingAsObject(0);
          String z = (String) bindings.getBindingAsObject(1);
          String y = (String) bindings.getBindingAsObject(2);
          if (true) { { } }
          urd.runtime.context.LocalBindings result_bindings =
            new urd.runtime.context.LocalBindings(2);
          result_bindings.addBinding(x);
          result_bindings.addBinding(z);
          return result_bindings;
      }
      catch (Exception e) {
          throw new urd.runtime.UrdRuntimeException(e);
      }
  }
```

Upon recognizing the context described by the exposingOrderedTracecut tracecut, the parser will call this method and provide it with an environment containing the values exposed by the primitive tracecuts in use. The first three boxed-off areas of this code show the assignment of this exposed context to the formal and semantic variables of exposingOrderedTracecut. The last two demonstrate that exposingOrderedTracecut exposes only x and z, passing a new environment containing these values back out to the parser. Since y is no longer needed it is left discarded.

The Constrained Order Tracecut Transformation A tracecut with semantic constraints is demonstrated next. Using the same assumptions as the previous exam-

ple, here we expose context, manipulate it, and conditionally reject the occurrence of this tracecut.

```
acecut constrainedOrderedTracecut(String x, String z) {: String y :}
::=  a(x) b(y) c(z) {:
    if(x.equals("bad")) { fail;}
      z = y;
;
```

As before, the same bindings are expressed by the tracecut. The semantic action block provides additional constraints on the acceptance of this tracecut. The resulting transformation is presented below.

```
ivate urd.runtime.context.IEnvironment
constrainedOrderedTracecut$R0(urd.runtime.context.IEnvironment bindings)
   throws urd.runtime.UrdRuntimeException {
      try {
          String x = (String) bindings.getBindingAsObject(0);
          String z = (String) bindings.getBindingAsObject(1);
          String y = (String) bindings.getBindingAsObject(2);
         if (true) {
              {
                  if (x.equals("bad")) { return null; }
                  z = y;
              }
          }
          urd.runtime.context.LocalBindings result_bindings =
            new urd.runtime.context.LocalBindings(2);
          result_bindings.addBinding(x);
          result_bindings.addBinding(z);
          return result_bindings;
      }
      catch (Exception e) {
          throw new urd.runtime.UrdRuntimeException(e);
      }
  }
```

With this example note that the semantic action placeholder has been populated

with the semantic action code provided by the tracecut. If the exposed string `x` is `"bad"` the method will return `null`. This `null` value has replaced `fail` (see inner nested box), and is used by the parser to signify failure (see Section 4.4.2). The `z` reference has been made to point to `y`, manually overriding the default bindings. The resulting bindings are again packed into a new environment and passed back to the parser as the return value.

This section has demonstrated how exposed context is both automatically and manually maintained in our resulting code. Methods for each clause in the grammar extracted for each tracecut are generated, and semantic action code is given a place to execute and conditionally reject the occurrence of these clauses in context.

4.4 Event Parsing

For each target tracecut, URD generates a context-recognizing parser that is then recombined with the aspect under transformation. These parsers are non-deterministic and employ a generalized parsing strategy permitting the parallel exploration of multiple interpretations of the observed context and allowing most specified context-free grammars to be used as a tracecut specification; the grammars are limited to those excluding right and hidden left recursion. In Section 4.4.1, we discuss how our parsers are generated from the extracted context-free grammar. In Section 4.4.2, we look at how they work to recognize context from the stream of events that are emitted as a result of the supporting transformations discussed in the previous section.

4.4.1 The Generation Process

Our parsers employ a context-free backbone in order to recognize the *unconstrained* execution contexts expressible by DEPs. To this backbone we add supporting mechanisms for exposing and constraining the recognition of execution context. The construction of this backbone follows the work of Aycock *et al.* [8, 7] and later Scott *et*

al. [77, 75, 74, 76]; these techniques aim to limit the use of parsing stacks by encoding as much left context information as possible in the automaton itself. Common approaches such as LL or LR parsers use a stack to maintain the context observed by the parser (see [2]). The alternate approaches we employ isolate the recursive nature of the grammar in question, and resort to the stack only when required. The resulting parsers consist of a hierarchy of regular automata. If recursion is present in the grammar, it is handled by calling into an appropriate regular sub-automaton. When parsing resumes from the sub-automaton, it returns at the context of the call. The approach mirrors subroutines in a programming language and employs a call stack in the same fashion.

An Example To demonstrate the nature of our context-free backbone, we generate a parser for the tracecut `isSafe` as though it were specified as target to a history application.

```
blic aspect GenerationExample {
// Ordered tracecut declarations
tracecut isSafe() ::= a() completed()* $;
tracecut completed() ::= a() [completed()] b()
                       | c() [completed()] d();

// Primitive tracecut declarations
tracecut a() ::=  entry(safePc());
tracecut b() ::=  exit(safePc());
tracecut c() ::=  entry(unsafePc());
tracecut d() ::=  exit(unsafePc());

// Pointcut declarations
pointcut safePc(): execution(* *.safe());
pointcut unsafePc(): execution(* *.unsafe(..));
```

The `isSafe` tracecut is a typical application of DEPs. Here we are interested in execution contexts where the current point in the execution of the program is more

tightly nested in executions of the `safe` method than the `isSafe` method. We use short names `a` and `b` to mark the entry and exit events of `safe` methods, and `c` and `d` for the entry and exit of `unsafe` methods. From this example, the following grammar is extracted. We abbreviate the longer tracecut names in the grammar. *Is* represents `isSafe`, *C* represents `completed`, and *Cs* refers to the rules that result from transforming `completed()*`. One can see that the optional tracecuts and Kleene star that were used in the tracecut specification have been expanded out.

$$
\begin{array}{lrcl}
\text{(Rule 0)} & S' & \rightarrow & Is\ \$ \\
\text{(Rule 1)} & Is & \rightarrow & a\ Cs \\
\text{(Rule 2)} & Is & \rightarrow & a \\
\text{(Rule 3)} & Cs & \rightarrow & Cs\ C \\
\text{(Rule 4)} & Cs & \rightarrow & C \\
\text{(Rule 5)} & C & \rightarrow & a\ C\ b \\
\text{(Rule 6)} & C & \rightarrow & c\ C\ d \\
\text{(Rule 7)} & C & \rightarrow & a\ b \\
\text{(Rule 8)} & C & \rightarrow & c\ d
\end{array}
$$

Derived grammar Construction begins with an analysis of the target grammar to find the points where actual recursion occurs. This analysis aims to derive a new grammar F that will specify the left context of each of the non-terminals from the original grammar G. The set of terminals in F include both the set of terminals and non-terminals of G. The non-terminals in F are derived (but not the same as those) from the non-terminals of G. These derived non-terminals are enclosed in square brackets to mark their significance (e.g., non-terminal $[A]$ derived from A).

The rules of the augmented grammar F are derived in the following manner. Firstly, a single rule is added $[S'] \rightarrow \epsilon$ to indicate that there is no left context for the root of the grammar. Intuitively, we want to build a grammar that describes all those strings of events that can come before each non-terminal in the original grammar. To this end, for each rule $[B] \rightarrow \alpha[A]\beta$ in G we add one rule $[A] \rightarrow [B]\alpha$ to F. The left context of $[A]$ is whatever can come before $[B]$ followed by α. Finally, F is made tidy

by removing direct cycles such as $[A] \rightarrow [A]$.

The derived grammar for our running example is shown in Figure 4.1. By con-

$$
\begin{array}{rcl}
[S'] & \rightarrow & \epsilon \\
[Is] & \rightarrow & [S'] \\
[Cs] & \rightarrow & [Is]\ a \\
[C] & \rightarrow & [Cs] \\
[C] & \rightarrow & [Cs]\ Cs \\
[C] & \rightarrow & [C]\ a \\
[C] & \rightarrow & [C]\ c
\end{array}
$$

Figure 4.1. *Derived Grammar for our running example*

struction the grammar F describes a regular (left-linear) language. Because F is a regular grammar, it is equivalent to a FA. The FA for F of our running example is shown in Figure 4.2. Finding recursive points in the original grammar is done by considering that the FA will contain a non-ϵ cycle if and only if the FA accepts an infinite language. By selecting a minimal set of transitions from the FA so that such cycles are then broken we can locate the points where recursion is exhibited. Aycock *et al.* point out that such a selection is equivalent to the feedback arc set (FAS) problem. While they point to heuristic algorithms to solve the problem, since URD is a proof-of-concept we simply remove all back-edges from the graph to break cycles. The FA for the derived grammar F of the running example is shown after removal of the feedback arcs in Figure 4.3.

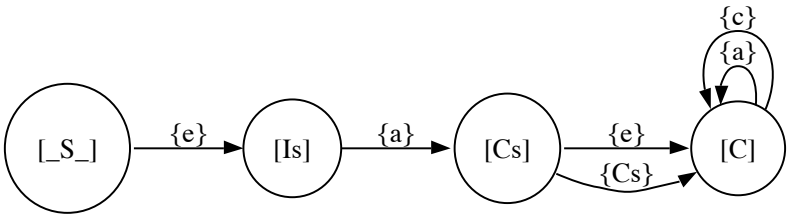

Figure 4.2. *Left context analysis*

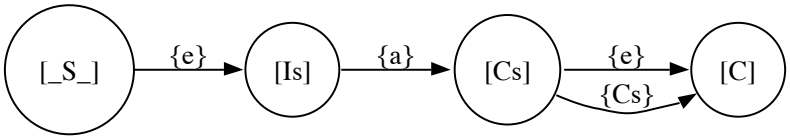

Figure 4.3. *Augmented Left context analysis. Recursion is removed.*

Limit points and Augmented grammar The broken cycles in the augmented grammar F (see Figures 4.2 and 4.3) are used to determine points in the original grammar G that lead to unavoidable recursion. Using Figure 4.3 as a guide, we can alter the original grammar to indicate these recursive points. These *limit points* in the grammar are so named because they indicate the points in the grammar where a finite memory reaches its limit and from then on a pushdown must be used.

We will call $\phi_{source} \xrightarrow{\alpha} \phi_{target}$ a transition between states *source* and *target* where α is some string of symbols labelling this transition. For each removed edge from the DFA in Figure 4.2, and for each ϕ_{source} production rule in G, mark occurrences of the label ϕ_{target} on the right-hand side of the rule with the special symbol lp, but only if preceded by the string of symbols α. In the case of our running example, the grammar in Figure 4.4 results.

$$
\begin{array}{lrcl}
\text{(Rule 0)} & S' & \rightarrow & Is\ \$ \\
\text{(Rule 1)} & Is & \rightarrow & a\ Cs \\
\text{(Rule 2)} & Is & \rightarrow & a \\
\text{(Rule 3)} & Cs & \rightarrow & Cs\ C \\
\text{(Rule 4)} & Cs & \rightarrow & C \\
\text{(Rule 5)} & C & \rightarrow & a\ lp.C\ b \\
\text{(Rule 6)} & C & \rightarrow & c\ lp.C\ d \\
\text{(Rule 7)} & C & \rightarrow & a\ b \\
\text{(Rule 8)} & C & \rightarrow & c\ d
\end{array}
$$

Figure 4.4. *Augmented Grammar for our running example*

Concretely, we have considered the edges $\phi_C \xrightarrow{a} \phi_C$ and $\phi_C \xrightarrow{c} \phi_C$ against rules 5, 6, 7, and 8, and found that C is proceeded by a in rule 5, and that C is proceeded by c in rule 6. Intuitively, only the points in the grammar where proper self-embedding

occurs are marked as limit points (Rules 5 and 6).

Viable Prefix (Φ) set Having now determined the recursive nature of the original grammar, and having marked such points in the grammar, we can now combine the limit-point grammar together with the augmented grammar (Figure 4.3) to generate the viable prefixes of each of the non-terminal symbols in the original grammar. Because cycles have been removed from Figure 4.3, the resulting generation of left-contexts is finitely large. The set of left-contexts combined with the *handles* (right-hand side of a grammar rule) of each non-terminal symbol (referred to here as the Φ set) is generated. Table 4.1 shows the left contexts and viable prefixes for the non-terminals in our running example.

Non-Terminal	Left context	Viable Prefix
S'	ϵ	$\epsilon\ Is\ \$$
Is	ϵ	$\epsilon\ a\ Cs$
		$\epsilon\ a$
Cs	$\epsilon\ a$	$\epsilon\ a\ Cs\ C$
		$\epsilon\ a\ C$
C	$\epsilon\ a\ Cs$	$\epsilon\ a\ Cs\ a\ lp.C\ b$
	$\epsilon\ a$	$\epsilon\ a\ a\ lp.C\ b$
		$\epsilon\ a\ Cs\ a\ b$
		$\epsilon\ a\ a\ b$
		$\epsilon\ a\ Cs\ c\ lp.C\ d$
		$\epsilon\ a\ c\ lp.C\ d$
		$\epsilon\ a\ Cs\ c\ d$
		$\epsilon\ a\ c\ d$

Table 4.1. Φ *set for our running example*

Parsing automata construction A trie is a tree data structure for storing a set of strings in which there is one node for every shared prefix. The construction uses a trie as the basis for the parsing automata. Those strings in the Φ set are added together to form a trie. Figure 4.5 shows the end result. With some effort one can see that the viable prefixes listed in Table 4.1 are fully represented in the trie structure.

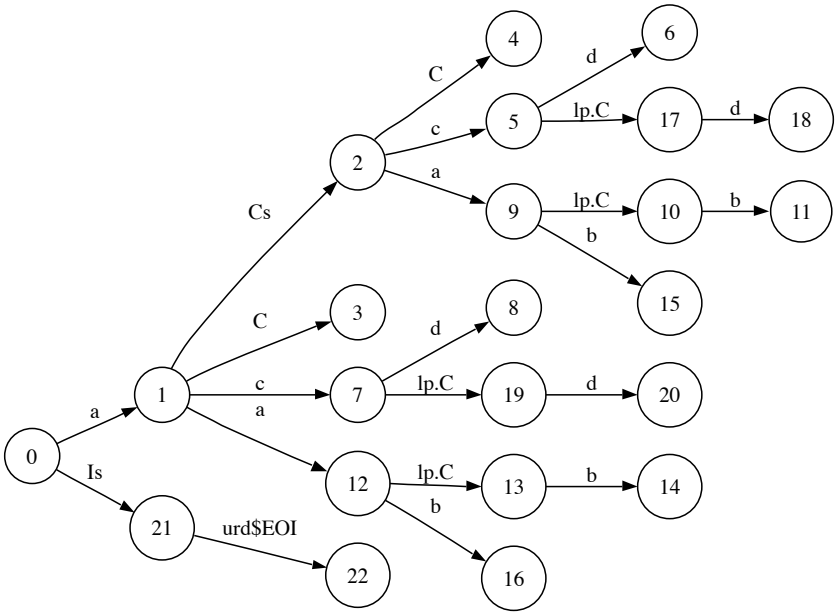

Figure 4.5. *Trie*

Special reduction edges are then added to the trie to recognize the occurrence of a given rule in the grammar. For each viable prefix, a reduction edge is added from the ending state of the prefix back to a state reachable by a transition on the non-terminal of the prefix, but only if such a state originates from a state reachable by tracing a handle of the non-terminal backwards from the end of prefix state. The reduction edge is then labelled to indicate the handle's corresponding production rule. For example, a reduction edge is added from state 20 to state 3, and is labelled REDUCE 5, because a backwards path from state 20 (the end state of a viable prefix of C) matches the handle of rule 5 ($C \rightarrow a\ lp.C\ b$).

Sub-automata The limit points in the grammar are those points where the use of the stack is unavoidable. Analogous to subroutines in a programming language, each unique limit point is expanding into a separate sub-automaton that matches a subset of the language. A new grammar is derived from the original grammar for each non-terminal represented by a limit point. This grammar contains those parts

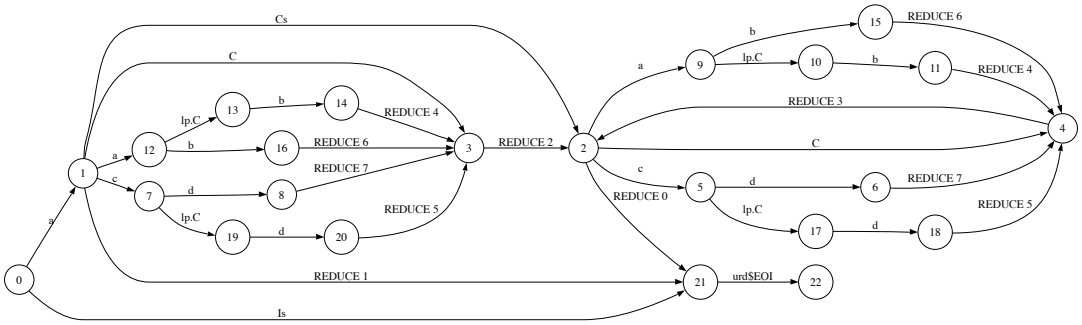

Figure 4.6. *Reduction Edges*

of the original grammar reachable from its corresponding non-terminal. From these new grammars, sub-automata are constructed through the same process described above. Figure 4.7 shows the automata for limit point C, as well as for our original target non-terminal Is.

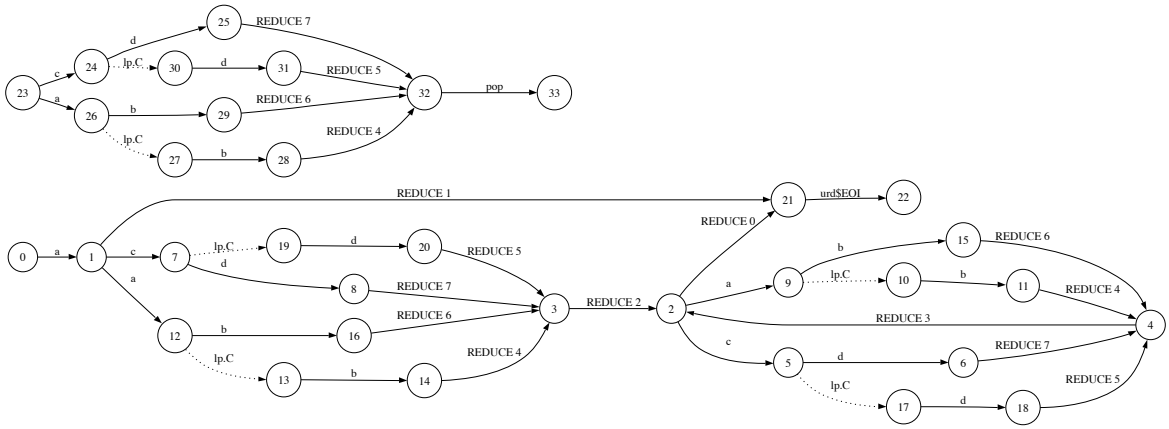

Figure 4.7. *Sub-automata*

Non-terminal edge removal and limit-point correction Once reduction edges are added, the non-terminal edges are removed, leaving a complete parsing automaton. The remaining limit point edges are converted to *calling* edges, resulting in a pushing of current state to the stack and resumption of recognition via a sub-automaton. *Returning* pop edges are added to return back to the calling context when the sub-automaton is complete. The final automaton is shown in Figure 4.8.

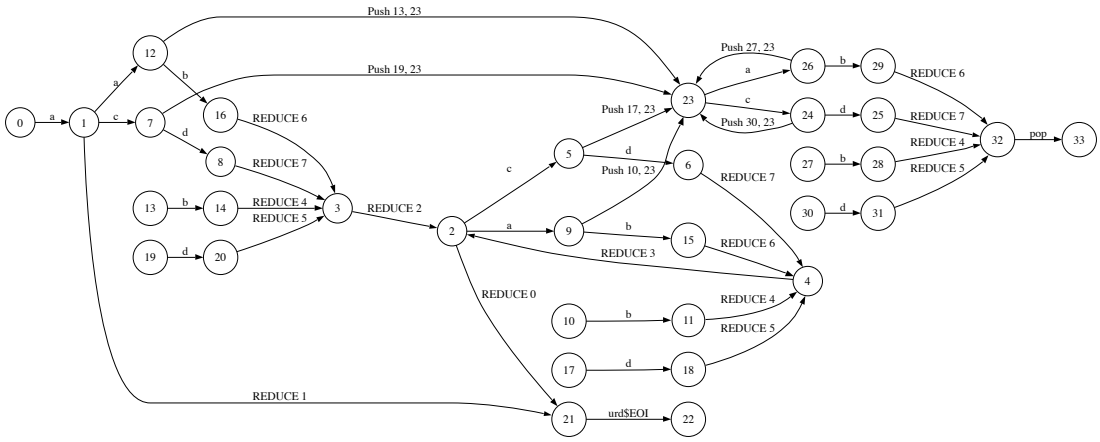

Figure 4.8. *Final automaton*

4.4.2 The URD Runtime

With the backbone in place, we now turn our attention to how it is traversed during runtime to recognize our contextual specifications. Our monitors employ a nondeterministic traversal permitting for the parallel interpretations of potentially ambiguous specifications. Potentially many *processing elements* are employed, each with its own parsing information, each exploring an alternate interpretation. In this section we begin with a description of the basic behaviours of our parsing elements. We then consider the implications of anchoring tracecuts, context exposure, failure and constraining variables.

Processing Elements The parsing backbone generated by our approach consists of states and edges. Each edge is one of a possible four types: *shift edges*, *reduce edges*, *push edges*, and *pop edges*. Our parsing elements independently traverse these edges in response to observed events in the execution trace.

1. *Shifting*: Shifting edges are followed in direct response to the observance of an expected event. They are labelled with the expected event type, and traversing the edge moves—or shifts—the processing element to the corresponding target state.

2. *Reducing*: Reduce edges are followed as part of the recognition of a rule in our tracecut grammar. Whenever the viable string of a given clause is recognized, these ϵ edges are followed; because clauses may consist of other ordered tracecuts, a chain of these ϵ edges are possible. In Figure 4.8 one can see reduction edges are labelled with the corresponding rule number. The methods generated for each clause in the grammar, which we discussed earlier, are executed as a result of traversing a reduction edge.
3. *Pushing*: Push edges are another form of ϵ edge. Pushing edges occur at those points in the recognition of a grammar where the finite state memory of our parsing automaton ends, and where a parsing stack must be employed. They correspond to the limit points of the grammar, and act to *call* on the sub-automaton. Pushing edges push the return state onto the parsing stack and then proceed to the target state.
4. *Popping*: Pop edges are analogous to returning from a subroutine. They are ϵ edges that are followed by popping the parse stack, and returning to the state that was popped.

Ambiguity and conflicts Ambiguous grammars may lead to situations where the processing element may have a choice of paths to follow. In deterministic parsers these sorts of conflicting choices must be disambiguated. In non-deterministic parsers such as ours, processing elements must split in two (or more) and follow each alternative path. This duplication process can be inefficient in that the entire parse stack of the duplicated processing element must be reproduced. By using the reduced stack-activity parsing generation techniques we have mentioned, a great deal of this inefficiency is mitigated. In the current version of URD we use a *tree-structured* parsing stack; duplicated processing elements share a common prefix. This is an improvement over complete stack replication, but more compact approaches are also available, such as the use of a *graph-structured* stack.

Anchors Anchors are a form of pattern constraint. Depending on the anchors used in a pattern, a traversal of the automaton may take different forms. Since there are two anchors, we have four possible traversal behaviours affecting when new processing elements are created (if they are created at all).

When a pattern is anchored to the beginning of the trace, there is no opportunity to consider new occurrences of a particular event. That is, there is no opportunity for the traversal to *reset*—no new generation of processing elements will be created when a *first* event occurs. The absence of the (^) anchor permits the pattern to be matched beginning at any point in the trace—this amounts to the introduction of a new processing element whenever an event in the first set of the target tracecut is observed.

The occurrence of the (**$**) anchor in a pattern indicates that, at any point when the parser is consulted, a processing element should be able to reach the end state of the automaton and provide a disambiguated reading. Processing elements are introduced to follow all transitions from current states to those that contain the (**$**) symbol. More than one of these processing elements may arrive at and converge on the final state. The question of which processing element should be presented as the parser's current match is implementation dependent—in order to maintain consistency (and to simplify the implementation) the last of the processing elements to arrive at the final state is kept. It should be emphasized that the parsing elements in place before this special exploration occurs remain in place and are ready to respond to the next event.

Whenever the (**$**) anchor is absent from a pattern, this indicates that the pattern may have matched in the past—the parser maintains the most recent of these matches. In order to ensure this most recent behaviour the process of traversing all ϵ edges is repeated. If the end state is reached, the most recent match is updated.

In the absence of a caret or dollar sign in a DEP, the DEP will be matched greedily from the beginning of the communication history. New processing elements will be

generated on the occurrence of first events, and (**$**) will be challenged after each normal processing round.

Context Exposure, Semantic Actions, and Binding Constraints When an exposed value is made available through the transformations shown for primitive tracecuts, this value is passed in a binding environment to the parser. Associated with each processing element is a shared *tree-structured* stack of environment frames. As context is exposed, it is placed in the top frame of this stack. Following an exposing shift edge corresponding to the first element of a potential handle causes a new binding frame to be added to the top of the processing elements binding environment. Reduction edges, after potentially manipulating this frame, result in the popping of the current frame.

During a reduction of a rule, the corresponding method generated for the rule is executed. As we saw, this method makes the variables in the current binding frame available to the body of this method. After the completion of the reduction method, a new environment frame containing possible modifications as a result of the semantic actions executed, or null is returned. If null is returned, that is an indication to the processing element that it should cease to exist. If an environment is returned, it is then combined with the new top frame through binding constraints. Binding constraints are just that; they dictate that exposed values are equivalent to other exposed values. Binding constraints are imposed on the the relationship between the positions in stacked environments.

4.5 Summary

In this chapter we explored URD, a prototype implementation of declarative event patterns. For each target tracecut URD extracts a context-free grammar as the basis for recognizing the execution contexts of interest. We discussed the general structure

of our tool, the way in which it transforms declarative event patterns aspects into regular AspectJ code, and how we generate and recombine the appropriate event monitors for each declarative event pattern. Additionally, we detailed the runtime behaviour of these monitors and the implications on performance. Our emphasis with URD has been on a practical implementation and not an optimized one. We point out the limitations of our current implementation and discuss future optimizations in Chapter 7.

Chapter 5

Validating Declarative Event Patterns

> Propose to an Englishman any principle, or any instrument, however admirable, and you will observe that the whole effort of the English mind is directed to find a difficulty, a defect, or an impossibility in it. If you speak to him of a machine for peeling a potato, he will pronounce it impossible: if you peel a potato with it before his eyes, he will declare it useless, because it will not slice a pineapple.
>
> —Charles Babbage (unconfirmed) [10]

The thesis of this work can be decomposed into two statements. The first such statement is that current aspect-oriented approaches fail to encapsulate context-sensitive crosscutting concerns in a manner that retains comprehensibility, traceability, and evolvability. The second claim of this thesis is that by using declarative event patterns the task of realizing context-sensitive crosscutting concerns is made easier, benefiting from improved comprehensibility, traceability, and evolvability that DEPs endeavour to provide. To validate these claims, we have performed two case studies involving the design and evolution of programs using both object-oriented and aspect-oriented approaches contrasted against declarative event patterns realizations. Each case study was designed to investigate specific research questions.

5.1 Methodology

The software engineering properties of traceability, comprehensibility and evolvability are difficult to quantify. The exploration of such phenomenon involves a large number of factors over which only a limited amount of control is available. The skill of the developer, their understanding of the technologies and techniques in use, their physical and emotional commitment to the task, the presence of environmental distractions all contribute. In our validation of declarative event patterns we have chosen to use the case study as our method of research; it is most suited to obtaining information which could explain qualitatively why our claims are valid or wrong.

5.2 Path-Specific Customization Study

In this case study we performed a path-specific customization on Columba using both Java, AspectJ and DEPs. This study has been used throughout this thesis to illustrate various facets of the problem we are trying to address and the solution we have arrived at. Recall that Columba is a open-source mail client written entirely in Java. It boasts a rich set of extensible features, some of which include support for IMAP, POP, message rule filters, junk filtering, internationalization and excellent interface responsiveness[1]. The participant for this case study (the author) had no previous exposure to the code of Columba. Path-specific customizations involve the modification or development of a program such as to adapt it to the requirements of a given execution context [19]. As we have demonstrated in Chapter 2, path-specific customization is achieved at the cost of modularity when using existing approaches.

[1]Details regarding Columba are available (at the time of writing) at http://columba.sourceforge.net/

5.2.1 Theory

The goal of the Columba study was mostly exploratory, to assess the practical benefits of using DEPs. The research questions motivating this case study were the following:

1. Can declarative event patterns adequately represent the intent of a developer when faced with the task of performing path-specific customizations?
2. What possible benefits result from declarative event patterns in this context?

Our initial theory for these questions was that:

1. DEPs should be able to adequately capture the intent of the programmer because they describe specific linear control sequences that mirror those found in the control flow of object-oriented programs, while disregarding details of such flow that are not essential to the concern.
2. DEPs support path-specific customization tasks by providing an uncluttered view of the program elements related to a particular path and of the relations between them, so that a developer can easily reason about, and effect the change.

5.2.2 Study Design

We perused Columba source code for candidate path-specific customization tasks. Columba (and in fact many mail clients) treat mail folders in a somewhat generic fashion. In particular, a common annoyance as a side effect of this is that, should a previously sent message (residing in the **Sent** folder) be replied to, the resulting composed message in effect is addressed to self. It is more often the case that one would want to continue on with additional points to the previously addressed recipient, not to themselves.

Initially we isolated the reply related behaviour of Columba as it pertains to composing message headers. In Chapter 2 we described how the selection of a message for reply within a given folder results in a triggered action, that in turn results in a

`ReplyCommand` object being constructed to facilitate the preparation and then creation of a message compose window containing the appropriate addressees.

A change task was formulated such that:

> Whenever a message residing in the **Sent** folder is replied to, regardless of the form of reply selected, the usual behaviour for collecting the `From` and `To` addresses should be supplanted with alternate behaviour to swaps the assignment of these fields, such that the reply acts as a continuation of the originally sent message. No other behaviour should be effected by this change.

Figure 5.1 shows a simplified picture containing the relevant portions of the reply system. Columba provides a number of reply types: `reply`, `reply-to-all`, `reply-to-mailinglist`, `reply-as-attachment`, and `reply-with-template`. In fact, the architecture of the reply mechanism is extensible via inheritance, offering in theory any number of new reply behaviours to be added. These reply subclasses may alter the behaviour of header assignment by overriding the `initHeader` method.

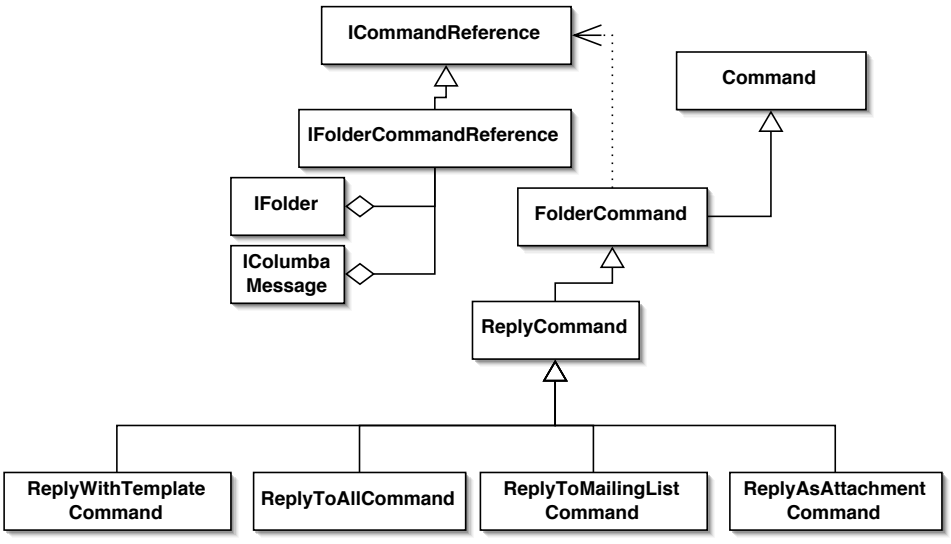

Figure 5.1. *Columba reply structure*

This study looked at how one would go about implementing the path-specification

customization using Java, AspectJ and DEPs. We summarize our findings for each of these approaches now.

***Ad hoc* Java Realization** An *ad hoc* Java implementation of this task was conducted initially to familiarize ourselves with the structure of Columba, and for later comparison against our AspectJ, and DEPs solution. The approach consisted of embedding code directly into `ReplyCommand` and its subclasses to ensure that the originating folder of the message under reply was accessible from each point where we required customization. That folder was then consulted to determine if it was in fact the **Sent** folder and then if the customized behaviour should be executed. The task was not difficult to do in pure Java, but exhibited a number of undesirable properties.

1. All subclasses of `ReplyCommand` that override `initHeader` required attention and duplication of the address swapping behaviour. Future `ReplyCommand` classes would require this treatment as well.
2. Context information regarding the folder from where the reply originated was passed down through to the reply composition resulting in extraneous dependency; folder information was required in methods for the sole purpose of realizing this customization. As it would happen, the developers of Columba pass this information down to support another concern, and so while the task of performing this change was made easier, it still exhibited these unsound observations. This can be seen in Figure 5.1; `ReplyCommand` extends `FolderCommand` which makes references, via a more general `ICommandReference`, to a realization of `IFolderCommandReference`, which is used to refer to the message and folder in which it originates.
3. For each subclass a conditional check was made before all calls to `Header.getFrom()`. If the folder in question was the **Sent**[2] folder, `Header.getFrom()` was condition-

[2]Columba provided an unnamed constant (`104`) representing the unique identifier of the **Sent** folder. The identifier of the current folder was compared against this value.

ally replaced with `Header.getTo()`. As a side-effect of the current Columba implementation, the resulting `from` field is populated according to the current account information, therefore no explicit change was needed.

Java and AspectJ Realizations In Chapter 2 we detailed a more structured Java realization that entailed instrumenting the base code in order to report the occurrences of context establishing events to a `Monitor` class. A related AspectJ realization of our path customization used advice to provide this instrumentation. In particular, both these implementations exhibited the following undesirable observations. In the case of the more structured Java approach:

1. All subclasses of `ReplyCommand` that override `initHeader` still required attention. In particular, event emitting instrumentation was scattered across in each of these modules. Future `ReplyCommand` classes would require this instrumentation as well.
2. Conditional checks were again made before all calls to `Header.getFrom()`. Instead of directly consulting the folder and establishing a further dependence on context as result, the state of the monitor class was consulted.
3. Recognition of context from primitive events required involved translation from intuitive trace-based specifications to finite state machines capturing the semantics of these traces, to state machine implementation in the form of a recognizer; the `Monitor` in the Java case.

In the case of the AspectJ approach:

1. The AspectJ approach alleviated points one and two above.
2. The AspectJ approach still exhibited the issues of point three above—recognition of contextual patterns largely remained in the hands of the developer.

DEPs Realization In considering how to customize Columba using declarative event patterns, we noted that an ideal solution should,

1. Address all existing and future extensions of `ReplyCommand` in a local encapsulated manner.
2. Should be insensitive to irrelevant code along the path of customization, permitting changes to occur irrespective of the general path assumed.
3. Should only apply to the context of interest (`Reply` commands for messages originating in the **Sent** folder).
4. Should be additive, in the sense that removing the path-specific behaviour should result in the behaviour of the existing system. In a nutshell, the path-specific customization should be pluggable.

The DEPs solution to our Columba customization presented in Chapter 3 achieves these goals. Figure 3.13 depicts a single modular aspect containing our customization concern. To address points one and two, we initially specified that we wanted to apply our customization only when the execution context conforms to the following pattern.

$$(\texttt{initHeader}^{call}_{exit})?\ (\texttt{getFolder}^{call}_{exit})\ (\texttt{initHeader}^{call}_{entry})\ (\texttt{getFrom}^{call}_{entry})^{+}\ \$$$

The DEPs equivalent that we use in our solution is as follows. The tracecut `customizationContext` captures our pattern quite directly.

```
nitHeaderExit()] sentFolderExit() initHeaderEntry() getFromEntry()+ $
```

This pattern is general enough such that any occurrences of `getFrom` within the dynamic extent of calls to `initHeader` that were initiated as a result of a reply to a previously-sent message will be replaced with `getTo`. Using primitive tracecuts to specify our context marking events, we are able to state those sets of join points across the entire system that match our needs. Future extensions providing such points in the program are caught by the general way we have stated these pointcuts. Our specification of context is specific to a small fragment of the reply path set out in the Columba system. This ensures that unless drastic refactoring of Columba is done, our tracecut will capture the desired context. If Columba were to undergo changes

such that this pattern no longer was correct, changing the tracecut would be nearly as straight-forward as changing the specification.

Our DEP-augmented aspect in Chapter 3 is self-contained and, because the base Columba implementation is oblivious to its presence due to the inversion of dependence provided by AspectJ, it is completely additive. DEPs provide a means to modify the semantics of existing code without invasively altering that code. The removal of the aspect depicted in figure 3.13 at build time would result in the original behaviour reaffirming itself.

5.2.3 Results

In order to evaluate our observations for this case study, we looked at how each of the approaches lead to more traceable, evolvable and understandable code. We found a gradation from *ad hoc* Java implementation through to DEPs solution. Along each dimension Java faired worse then AspectJ, and AspectJ worse then DEPs.

Invasive Modifications The Java implementations involved invasive modification of multiple class modules in order instrument context establishing points and to perform checks against the context. While these changes were simple in nature, this sort of scattering has been linked with difficulty in tracing, evolving and understanding a concern.

The AspectJ solution (as well as the DEPs solution) alleviated the need for invasive modification. Due to the nature of AOP, both the AspectJ and DEPs aspects were woven into the base reply mechanisms avoiding the dependencies otherwise established in the Java approaches. Further to this, both with both AspectJ and DEPs the contextual marker event declarations were localized within a single aspect, along with the context recognition portions of each solution.

Context recognition In the *ad hoc* Java implementation, no means for context recognition is required at runtime. Instead, there is a direct dependence on the structure and dynamic flow of control of the Columba reply mechanism. Code was inserted directly along existing control paths that would lead to the correct context. This approach is fragile at best. It assumes that the structure of the Columba reply mechanism will not evolve. If this structure did change, perhaps refactored in such a way to reorganize the code, but without interfering with the general flow of control, this would invalidate the embedded code of this approach. Understanding this code requires a complete consideration of all affected modules. Traceability is lost; the customization exhibits itself in various separate modules and does not resemble the original specification.

In Chapter 2 we demonstrated the problems with a more methodical Java approach which used instrumentation and a monitor class to recognize context. From our trace-based specification, a complete state-based model was derived, which then was translated into code. This staged process widens the gap from specification to implementation. It was no longer possible to look at the code and directly determine what the context of interest was. Understanding the implementation requires a careful consideration to reconstruct the state machine, and then make sense of that state machine in terms of the context it recognizes. Changing the context specification would involve a repeat of the entire process.

The AspectJ approach, save for allowing event instrumentation to be locally declared and realized in a non-invasive manner, exhibited precisely the same problem seen in our methodical Java approach.

With DEPs, we were able to take our regular expression specification of context and capture it directly in code. The declarative nature of DEPs allowed us to state the pattern just as we originally conceived it—as a description of those paths necessary for establishing context. As a result, comparing our tracecut implementation to the original specification is easy. There is a direct correspondence. Changes to the spec-

ification mean similar changes to the tracecut. If one understands the specification, they understand the corresponding tracecut.

Summary The research questions we set out to answer in this study appear to have been addressed in the affirmative.

1. DEPs was able to adequately capture the intent of the programmer by mirroring the specification of contextual requirements closely while disregarding details of such flow that are not essential to the concern. The correspondence between the control flow of object-oriented systems and the languages described by context free grammars allows us to specify this regular path without trouble.
2. The path-specific customization using DEPs was an improvement over existing approaches in terms of comprehension, traceability, and evolvablilty. Our specification of context was highly traceable with the DEPs implementation. There is an immediate correspondence between our specification of context, and the tracecut realization of that specification. Because of this correspondence, understanding and changing the tracecut was improved over the object-oriented and aspect-oriented approaches.

5.3 File Transfer Protocol Study

In the FTP case study, the author of this thesis took the role of a maintenance programmer to extend and then evolve a custom FTP server implementation with features for authenticating users. This extension was implemented four different times; each time an alternate approach was applied. The first approach used pure object-oriented techniques. The same extension was then implemented using AspectJ, using current aspect-oriented techniques. A third attempt at implementing authentication was made using the techniques and tool for Event-based AOP (EAOP). Finally, our DEPs tool was deployed to implement and evolve authentication.

The File Transfer Protocol (FTP) defines, among other details: (1) a set of commands that a client may send to a server, (2) a procedure for authenticating the user who is interacting with the client, and (3) the effects of authentication or lack thereof on the remainder of the functionality of the protocol.

User authentication involves two FTP commands issued by a client. The `USER` command passes an argument that supplies the user name to the server. The `PASS` command passes an argument that supplies the user's password to the server. The FTP specification (RFC 959 [68]) makes two statements regarding the sequencing of these and other FTP commands:

> [The `PASS`] command must be immediately preceded by the user name command.
>
> Servers may allow a new `USER` command to be entered at any point .... This has the effect of flushing any user, [and] password ... information already supplied and beginning the login sequence again.

The session is authenticated at a particular moment if and only if the most recent occurrence of the `USER` command is immediately followed by a `PASS` command, this `PASS` command is the most recently issued, and the password supplied in that `PASS` command is valid for the user name supplied in that `USER` command. This is a simple pattern match on the execution trace that can be expressed as the regular expression below (via `grep` syntax); a password check must be performed in addition.

```
USER PASS [^USER,PASS]* $
```

The finite state machine for user authentication is shown in Figure 5.2; it can be derived through careful analysis of the FTP specification and corresponds to the regular expression above. This state machine must possess three states: **U**nauthenticated, awaiting **P**assword, and **A**uthenticated. The state machine begins in state U. Receipt of a `USER` command causes a transition to state P regardless of the current state. Receipt of a `PASS` command in states U or A causes a transition to state U.

A transition to state U also occurs if the password contained in the `PASS` command is invalid. However, if a `PASS` command is received in state P that contains a valid password, the server transitions to state A. And finally, receipt of any other FTP command while in state P causes the system to revert back to state U. Receipt of other FTP commands in states U and A cause no change.

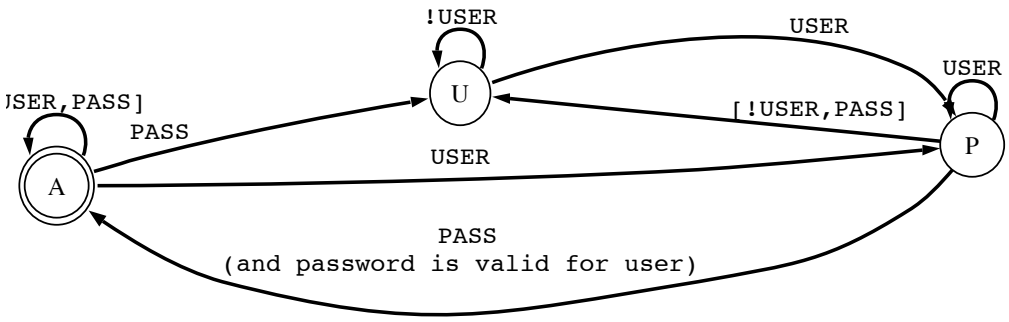

Figure 5.2. *FSM for authentication implied by RFC 959*

5.3.1 Theory

The primary goal of the FTP study was to evaluate the practical benefits of using DEPs when faced with the evolution of a non-trivial system. The research questions motivating this case study were the following:

1. Can declarative event patterns adequately represent the intent of a developer when faced with the task of enforcing sequential authentication policy?
2. How do DEPs compare with existing approaches for performing such tasks? What benefits, if any, do DEPs have over such approaches?

Our initial theory for these questions was that:

1. DEPs should be able to adequately capture the finite state authentication policy (as seen in Figure 5.2), and affect the required semantic changes to the FTP servers response depending on the execution events that have transpired.

2. Compared to the other approaches (OO, AOP, EAOP), DEPs should exhibit better comprehensibility, traceability, and evolvability of the FTP authentication policy, allowing for such policies to be more easily amended and alternative policies to be explored.

5.3.2 Study Design

Initially, we designed and implemented, in Java, an FTP server providing the Minimum Implementation subset of features as specified by RFC 959. A simplified UML class diagram for this design is shown in Figure 5.3. This base design ignored the presence of the remainder of the FTP specification. Each FTP command was implemented in a separate subclass (via the Command design pattern); we do not describe the base design further. The base design was then extended in four versions to add the user authentication feature, one each via Java, AspectJ, EAOPTool, and our proof-of-concept tool for DEPs. We then evolved each system to alter the authentication feature to account for a special case in the FTP specification. We compare the resulting implementations for their effects on traceability, comprehensibility, and evolvability.

FTP Authentication in Java Authentication required additions to several classes in the base design. A `PasswordCommand` was added to the `Command` hierarchy to parse the arguments to the `PASS` command. The `UserCommand` had to be altered to reply "Password needed" to the client. An `AuthenticationMonitor` class was added to monitor and record those events involved in determining the current authorization state of a `Session`. (`AuthenticationMonitor` combines elements from the State and Mediator design patterns.)

Since the occurrence of any FTP command can effect a state transition within the monitor in some states, every class in the `Command` hierarchy had to be modified to notify the monitor of the occurrence of an FTP command event. The `Password-`

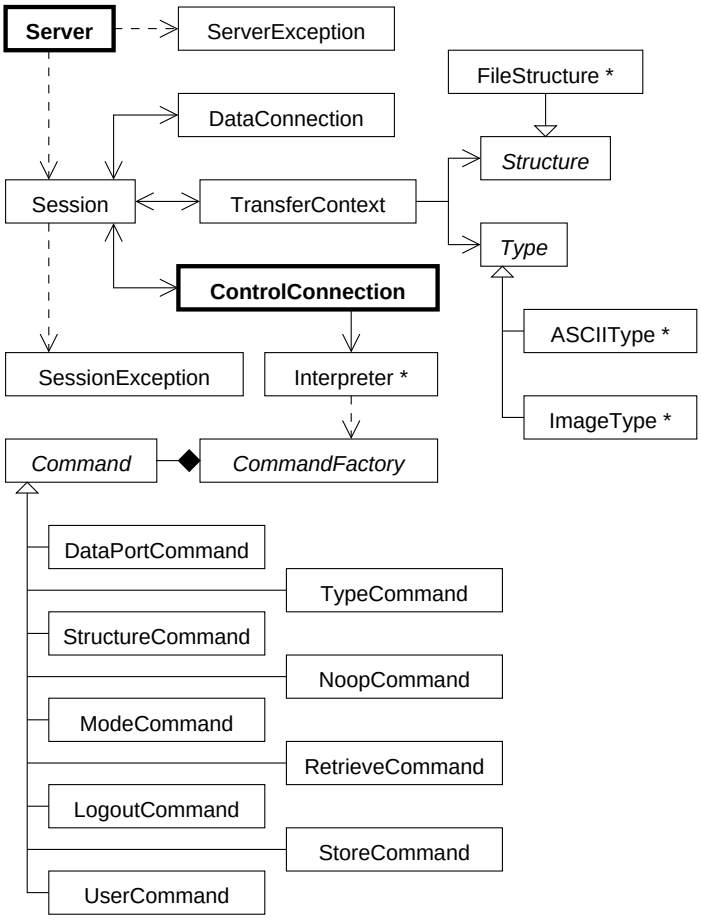

Figure 5.3. *Base design for the FTP server.*

`Command` class was instrumented to report occurrences of `PASS` events, `UserCommand` reported `USER` event occurrences, and all other `Command` subclasses reported `OTHER` event occurrences.

Figure 5.4 shows the implemented `AuthenticationMonitor` class. Anonymous classes are used to represent the three authentication states described earlier. The monitor is initially in state U. Each state class must implement a method to react to each of the three recognized event kinds. The implementation will cause a transition to another state when appropriate.

In addition to reporting the occurrence of events, most FTP commands (except `USER`, `PASS`, or `QUIT`) must be authenticated prior to operation. Therefore, six of the nine subclasses in the `Command` hierarchy had to be further altered to check that the user had been logged in. If not, a reply of "User not logged in" had to be sent to the client. For most commands, this involved the insertion of a few lines of code at the beginning of the corresponding `perform` method on that `Command` subclass; this code calls the `isAuthenticated` method of `AuthenticationMonitor`.

FTP Authentication in AspectJ The AspectJ extension (Figure 5.5) followed an approach to user authentication similar to that in the Java extension. A single aspect named `AuthenticatorMonitor` and a Java class named `PasswordCommand` equivalent to the one added for the Java extension were created.

Rather than manually instrumenting each `Command` subclass to report particular event occurrences to the `AuthenticationMonitor`, we were able to use AspectJ to instrument the corresponding join points in a generative fashion. Four named pointcuts were declared to capture occurrences of FTP commands: one for `USER`, one for `PASS`, one for `QUIT`, and one for all commands except `USER` or `PASS` (`other`); an additional named pointcut (`needAuthentication`) captures join points where user authentication is required. Three pieces of `before` advice were declared to instrument the base code. Each of these notifies an `AuthenticationMonitor` state machine that an event

```
1 public class AuthenticationMonitor {
2
3   /* State and transition definitions */
4   private abstract class AuthenticationState {
5     public void observePasswordCommand(String pwd) {}
6     public void observeUserCommand(String usr) {}
7     public void observeOtherCommand() {}
8   }
9
10  AuthenticationState unauthorizedState = new AuthenticationState() {
11    public void observeUserCommand(String usr) {
12      state = pendingState;
13      userName = usr;
14    }
15  };
16
17  AuthenticationState authorizedState = new AuthenticationState() {
18    public void observePasswordCommand(String pwd) {
19      state = unauthorizedState;
20      userName = null;
21    }
22    public void observeUserCommand(String usr) {
23      state = pendingState;
24      userName = usr;
25    }
26  };
27
28  AuthenticationState pendingState = new AuthenticationState() {
29    public void observePasswordCommand(String pwd) {
30      if(Authenticator.isValid(userName, pwd)) {
31        state = authorizedState;
32      } else {
33        state = unauthorizedState;
34      }
35    }
36    public void observeOtherCommand() {
37      state = unauthorizedState;
38    }
39    public void observeUserCommand(String usr) {
40      userName = usr;
41    }
42  };
43
44  /* Protocol context */
45  private AuthenticationState state = unauthorizedState;
46  private String userName = null;
47  /* Mediation method */
48  public boolean isAuthenticated() {
49    return (state == authorizedState);
50  }
51  /* Observation methods */
52  public void observeOtherCommand() {
53    state.observeOtherCommand();
54  }
55  public void observeUserCommand(String usr) {
56    state.observeUserCommand(usr);
57  }
58
59  public void observePasswordCommand(String pwd) {
60    state.observePasswordCommand(pwd);
61  }
62 }
```

Figure 5.4. *Partial authentication extension in Java*

```
1 public aspect AuthenticationMonitor {
2   /* State and transition definitions, protocol
3      context, and mediation method all identical
4      to Java extension, so not repeated... */
5
6   /* Observation pointcuts */
7   pointcut user(String usr):
8     execution(*
9       UserCommand.perform(String, Session))
10    && args(usr);
11
12  pointcut pass(String pwd):
13    execution(*
14      PasswordCommand.perform(String, Session))
15    && args(pwd);
16
17  pointcut other():
18    execution(* Command+.perform(..))
19    && !user(*) && !pass(*);
20
21  pointcut quit():
22    execution(*
23      QuitCommand.perform(String, Session));
24
25  pointcut needAuthentication(Session s):
26     other() && !quit() && args(s);
27
28  /* Behaviour modification */
29  void around(Reply r):
30    cflow(user(String))
31    && args(r)
32    && call(* * ControlConnection.send(Reply)) {
33      proceed(new NeedPasswordReply());
34  }
35
36  void around(Session s): needAuthentication(s) {
37    if(!isAuthenticated) {
38      ControlConnection cc =  s.getControlConnection();
39      cc.send(new NotLoggedInReply());
40    } else {
41      proceed(s);
42    }
43  }
44
45  void around(Reply r):
46    cflow(pass(*)) && args(r) &&
47    call(* * ControlConnection.send(Reply)) {
48    if(!isAuthenticated()) {
49      proceed(new LogInFailedReply());
50    } else {
51      proceed(r);
52    }
53  }
54  /* Observation instrumentation */
55  before(String usr): user(*) && args(usr) {
56    state.observeUserCommand(usr);
57  }
58  before(String pwd): pass(*) && args(pwd) {
59    state.observePasswordCommand(pwd);
60  }
61  before(): other() && !quit() {
62    state.observeOtherCommand();
63  }
64 }
```

Figure 5.5. *Partial authentication extension in AspectJ*

of potential interest has occurred; the monitor was implemented identically to that in the Java extension.

Three pieces of around advice were declared to alter the response of a `Command` based on the current `AuthenticationState`. The first of these alters the behaviour of `UserCommand`: the receipt of `USER` causes a "Password needed" reply to be sent. The second advice alters invalid command executions so that a "Not logged in" reply is sent instead of servicing the request. The third advice captures a dummy reply issuing from `PasswordCommand` as a convenient join point to add the following functionality. Within the Pending state, if an invalid password is received as the argument to a `PASS` command, a reply to this effect must be sent.

FTP Authentication in Event-Based AOP Event-based AOP (EAOP) [26, 27] monitors the occurrence of particular events in the execution of a system. The developer specifies a set of join points in the source code of a program. When these points are reached, events are emitted to an event monitor that operates as a co-routine to the main program. The monitor passes these events to developer-defined subclasses of a library class called `Aspect`. Each subclass must be implemented to parse the incoming stream of events to recognize some pattern of interest. When a pattern is recognized, developer-specified behaviour occurs. As a result, the equivalent of AspectJ advice may be applied to complex sequences of events.

Part of the implementation of the FTP user authentication extension in EAOP is shown in Figure 5.6. The developer must define the points in the base functionality that are to emit events when they are executed. Library and tool support for this instrumentation process has been added to the EAOP tool. We do not show the code that specifies the instrumentation points; it is conceptually equivalent to the AspectJ pointcut definitions described earlier.

We added an `AuthenticationMonitor` subclass to monitor and to capture events in the base FTP functionality. In order to recognize the pattern of events related to

```
class AuthenticationMonitor extends Aspect {
  /* Protocol context */
  boolean isAuthenticated = false;

  /* EAOP Aspect entry point */
  public void definition() {
    MethodCall userCall = null;
    MethodCall passCall = null;
    Event e = null;
    while(true) {
      e = nextCallEvent();
      while(!isUserCommand(e)) {
        e = nextCallEvent();
      }
      while(isUserCommand(e)) {
        userCall = (MethodCall)e;
        e = nextCallEvent();
      }
      if(isPasswordCommand(e) && userCall != null) {
        passCall = (MethodCall)e;
        String usr = (String)userCall.args[0];
        String pwd = (String)((MethodCall)passCall).args[0];
        if(Authenticator.isValid(usr, pwd)) {
          isAuthenticated = true;
        } else {
          isAuthenticated = false;
          passCall = null;
          userCall = null;
        }
      } else {
        isAuthenticated = false;
        passCall = null;
        userCall = null;
      }
      // Some other details elided
    }
  }

  /* Event classification methods */
  public boolean isPasswordCommand(Event e) {
    return ((e instanceof MethodCall) &&
            ((MethodCall) e).method.getDeclaringClass().getName().equals("PasswordCommand"));
  }
  public boolean isUserCommand(Event e) {
    return ((e instanceof MethodCall) &&
            ((MethodCall) e).method.getDeclaringClass().getName().equals("UserCommand"));
  }
  public boolean isCallEvent(Event e) {
    return (e instanceof MethodCall);
  }

  /* Block on next event and filter out non-call events */
  public MethodCall nextCallEvent() {
    Event e = null;
    boolean ok = false;
    while(!ok) {
      e = nextEvent();
      ok = isCallEvent(e);
    };
    return (MethodCall)e;
  }
}

// Event emitter instrumentation not shown but conceptually equivalent to the AspectJ pointcuts
// Action language specification not shown but conceptually equivalent to the AspectJ behavioural advice
```

Figure 5.6. *Partial authentication extension in EAOP*

user authentication, the developer must provide a method called `definition` within the subclass that parses the events. With some difficulty, one can identify that the necessary authentication sequence through the finite state machine of Figure 5.2 is implemented in this method. The state is recorded in the pair of local variables `userCall` and `passCall`: when both are `null`, the state is **U**nauthenticated; when `userCall` references an object, the state is awaiting **P**assword; and when both local variables contain object references, the state is "ready to be authenticated" as the actual password check must be performed.

To modify the behaviour of the base program according to the authentication state, the developer must provide composition specifications that are conceptually equivalent to AspectJ advice declarations. We found that the EAOPTool had a number of shortcomings in terms of the usability of its composition specifications; however, these shortcomings could presumably be overcome with additional development effort. As these details are not pertinent to our discussion and would clutter the code snippet significantly, we do not show them.

FTP Authentication via DEPs The application of DEPs towards the implementation of the FTP authentication protocol required the addition of a single DEP-augmented aspect to the system, shown in Figure 5.7. The five pointcuts we used are identical to those found in the original AspectJ aspect of Figure 5.5. For brevity we do not reproduce them here.

In place of the combined state machine implementation and advice for observation instrumentation, we provide a single tracecut. This tracecut declares two local variables, `name` and `pwd`. The tracecut matches occurrences of the pattern "`user pass other`*" where the passed user name and password are an authenticated pair. This latter test is specified as a semantic action block; if it fails, the parser is informed that the event sequence does not match and that the start state should be re-entered, ready for new events. The "`$`" matches the current end of the trace prefix.

```
1
2 public aspect AuthenticationMonitor {
3   /* Observation pointcuts */
4   tracecut user(String usr): entry(user(usr));
5   tracecut pass(String pwd): entry(pass(pwd));
6   tracecut other(): entry(other());
7   pointcut quit(): execution(* QuitCommand.perform(String, Session));
8   pointcut needAuthentication(Session s):  other() && !quit() && args(s);
9
10  /* Event pattern detection */
11  tracecut isAuthenticated() {: String name, String pwd :} ::=
12    ( user(name) pass(pwd) {:  if(!Authenticator.isValid(name, pwd)) fail;:}) other()* $;
13
14  /* Behaviour modification */
15  void around(Session s): needAuthentication(s) && !history(isAuthenticated()) {
16     ControlConnection cc = s.getControlConnection();
17     control.send(new NotLoggedInReply());
18  }
19
20  void around(Reply r):
21    cflow(pass(*)) &&
22    args(r) &&
23    call(* * ControlConnection.send(Reply)) &&
24    !history(isAuthenticated()) {
25        proceed(new LogInFailedReply());
26  }
27 }
28
```

Figure 5.7. *Full authentication extension using DEPs*

Finally, there are two pieces of `around` advice that correspond closely to those in the original AspectJ aspect. The only difference is that these ones make use of the tracecut within a `history` pointcut instead of the `if` statements in the original. The `history` pointcut matches all join points where the DEP specified as an argument matches the current trace prefix.

Evolving Authentication As a final task, the FTP authentication policy was altered slightly in order to accommodate for the ACCT FTP command (RFC 959 [68]) used on systems that distinguish between user accounts, and user names. This command is intended to support systems requiring account information in addition to user information when performing certain commands. An FTP server might be configured in either of following two ways:

C1 The underlying system requires account information in addition to the normal

login sequence (USER, PASS). The ACCT command (with string argument) is expected to immediately follow a successfully PASS command (as indicated to the client with a unique return code).

C2 The underlying system does not require account information for login, but does requires account information for purposes of other commands (such as file storage), and these other commands can potentially occur at any point in an authenticated session dialog; the NEED-ACCT reply should be sent in response to such commands unless the underlying FTP server implementation maintains previously seen account information as part of the session state, in which case the previous information is used.

In both cases, voluntary `ACCT` commands may occur at any point during a validated session, changing the current account information if the account corresponds to the user.

Using Java, AspectJ, and DEPs we explored each server configuration. For Configuration 1 (**C1**), the authentication pattern used earlier in this study was modified as follows.

```
USER PASS ACCT (ACCT|[^USER,PASS])* $
```

In addition to the occurrence of `USER` and `PASS`, this configuration requires that an `ACCT` message come immediately after `PASS`; it also permits the occurrences of additional `ACCT` messages to allow for voluntary changes to the account information.

With Configuration 2 (**C2**), the basic authentication specification is still necessary; `ACCT` information may be required or volunteered at any point during an authenticated session. We use two context specifications to capture this. The first specification is simply the same authentication pattern we used previously in this case study. The second pattern captures the most recent occurrence of `ACCT`, but only if the session is authenticated.

```
USER PASS ([^USER,PASS])* $
```

```
([USER,PASS])* (ACCT)+ $
```

State-machine Evolution For the Java and AspectJ approaches we needed to derive formal state machines suitable for capturing the new contextual requirements. Figure 5.8 depicts the derived finite state machines for user authentication including ACCT commands.

C1 For Configuration 1 we have added an addition state "Pending ACCT" to account for the additional stage in the authentication policy. This can be seen in the right-most FSM in Figure 5.8.

C2 For Configuration 2 we utilize a combination of the original finite state machine from Figure 5.8 (left FSM), and a new one for capturing the most recent occurrence of `ACCT` (middle FSM). This one used in conjunction with the original authentication FSM to determine the currently valid account information.

With state machines determined, we followed the established implementation approach. With both configurations we added to our FTP command hierarchy an `AcctCommand` subclass. Instrumentation, context checks and context recognition were realized. Instrumentation (or event-emitting advice in the case of AspectJ) to report the occurrences of `ACCT` commands to the monitor (or aspect in the case of AspectJ) was added. In the case of Java, commands requiring `ACCT` information were altered to consult the state of the monitor and to reply with the `ACCT-NEEDED` reply if authentication was not properly established. Using AspectJ, this was achieved through advice at those points, rather than explicit scattered instrumentation. We focus on the recognition in the remainder of this section. The two resulting state-based implementations are seen in Figures 5.9, 5.10 and 5.11.

C1 Figures 5.9 and 5.10 capture the context recognition required for Configuration 1. This state-machine was translated into Java code in a similar manner to the object-oriented and aspect-oriented approach we used for the original

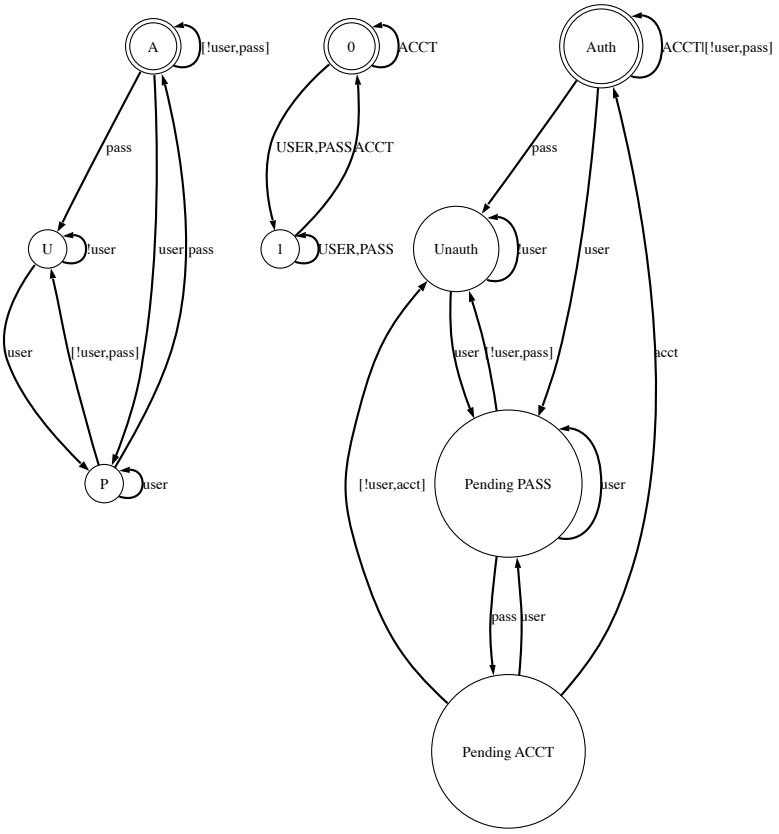

Figure 5.8. *Finite state machines for different ACCT scenarios*

authentication specification. In these figures a single new `Authentication-Monitor_Config1` class is defined. In incorporates into the original `Authentication-Monitor` a new state `pendingAcctState`, and a number of new and altered transitions on the observance of `AcctCommand`. The majority of the original `AuthenticationMonitor` was altered to account for the change in requirements. Each addition or alteration is boxed in these listings to emphasize the impact of this evolution.

C2 Figures 5.11 lists the new `AuthenticationMonitor_Config2` class. For Configuration 2 the existing recognizer was coupled with an additional recognizer to capture the most recent occurrence of the `ACCT` information. Again, `ACCT` occurrences were instrumented and fed to the new recognizer. However, an additional filtering step was employed (lines 46-50). If the session was authenticated, knowledge of the occurrence of `ACCT` would proceed on to the second recognizer. If the session was not authenticated, the `ACCT` occurrence would be ignored. Additionally, instrumentation (or advice) to report occurrences of `USER` and `PASS` where added to report to the second recognizer; if the session becomes unauthenticated, the most recent account information must be discarded (lines 20 and 27). In this listing the contexts of our orginal `AuthenticationMonitor` are assumed. When we make reference to part of that assumed code, it is boxed (line 47).

Declarative Event Pattern approach Using DEPs we were able to extend our solution in a straight-forward manner. Building off of our existing DEPs solution, Figures 5.12 and 5.13 show our evolution for Configuration 1 and Configuration 2 respectively.

C1 For Configuration 1, we are interested in capturing

```
USER PASS ACCT (ACCT|[^USER,PASS])* $
```

```
1 public class AuthenticationMonitor_Config1 {
2   /* State and transition definitions */
3   private abstract class AuthenticationState {
4     public void observePasswordCommand(String pwd) {}
5     public void observeUserCommand(String usr) {}
6     public void observeOtherCommand() {}
7     public void observeAcctCommand(String acct) {}
8   }
9
10  AuthenticationState pendingAcctState = new AuthenticationState() {
11    public void observeUserCommand(String usr) {
12      state = pendingPassState; userName = usr;
13    }
14    public void observeAcctCommand(String acct) {
15      if(Authenticator.isValidAcct(userName, acct)) {
16        state = authorizedState; acctName = acct;
17      } else {
18        state = unauthorizedState;
19      }
20    }
21    public void observeOtherCommand() {
22      state = unauthorizedState;
23    }
24    public void observePassCommand(String pass) {
25      state = unauthorizedState;
26    }
27  };
28
29  AuthenticationState authorizedState = new AuthenticationState() {
30    public void observePasswordCommand(String pwd) {
31      state = unauthorizedState; userName = null;
32    }
33    public void observeUserCommand(String usr) {
34      state = pendingPassState; userName = usr;
35    }
36    public void observeAcctCommand(String acct) {
37      if(Authenticator.isValidAcct(userName, acct)) {
38        acctName = acct;
39      }
40    }
41  };
```

Figure 5.9. *This figure shows part 1 of the evolution of the FTP authentication recognizer according to the requirements of configuration 1.*

```
1  AuthenticationState unauthorizedState = new AuthenticationState() {
2    public void observeUserCommand(String usr) {
3      state = pendingPassState; userName = usr;
4    }
5  };
6
7  AuthenticationState pendingPassState = new AuthenticationState() {
8    public void observePasswordCommand(String pwd) {
9      if(Authenticator.isValid(userName, pwd)) {
10       state = pendingAcctState;
11     } else {
12       state = unauthorizedState;
13     }
14   }
15   public void observeOtherCommand() {
16     state = unauthorizedState;
17   }
18   public void observeAcctCommand(String acct) {
19     state = unauthorizedState;
20   }
21   public void observeUserCommand(String usr) {
22     userName = usr;
23   }
24 };
25
26 /* Protocol context */
27 private AuthenticationState state = unauthorizedState;
28 private String userName = null;
29 private String acctName = null;
30
31 /* Account accessor method */
32 public String getAcctName() { return acctName; }
33
34
35 /* Mediation method */
36 public boolean isAuthenticated() {
37   return (state == authorizedState);
38 }
39 /* Observation methods */
40 public void observeOtherCommand() {
41   state.observeOtherCommand();
42 }
43 public void observeUserCommand(String usr) {
44   state.observeUserCommand(usr);
45 }
46 public void observePasswordCommand(String pwd) {
47   state.observePasswordCommand(pwd);
48 }
49 public void observeAcctCommand(String acct) {
50   state.observeAcctCommand(acct);
51 }
52 }
```

Figure 5.10. *This figure shows part 2 of the evolution of the FTP authentication recognizer according to the requirements of configuration 1.*

```
1 public class AuthenticationMonitor_Config2 {
2
3   /* The AuthenticationMonitor declarations of Figure 5.4 are assumed but not shown. */
4
5   private abstract class MostRecentAcctState {
6     public void observePasswordCommand(String pwd) {}
7     public void observeUserCommand(String usr) {}
8     public void observeAcctCommand(String acct) {}
9   }
10
11  MostRecentAcctState invalidAcctState = new MostRecentAcctState()    {
12    public void observeAcctCommand(String acct) {
13        acctName = acct;
14        state = validAcctState;
15    }
16  };
17
18  MostRecentAcctState validAcctState = new MostRecentAcctState()    {
19    public void observePasswordCommand(String pwd) {
20        acctName = null;
21        state = invalidAcctState;
22    }
23    public void observeAcctCommand(String acct) {
24        acctName = acct;
25    }
26    public void observeUserCommand(String usr) {
27        acctName = null;
28        state = invalidAcctState;
29    }
30  };
31
32  /* Protocol context */
33  private MostRecentAcctState state = invalidAcctState;
34  private String acctName = null;
35  /* Mediation method */
36
37  public boolean isAcctSet() {
38    return (state == validAcctState);
39  }
40
41  public String getAcctName() {
42    return acctName;
43  }
44
45  /* Observation methods */
46  public void observeAcctCommand(String acct) {
47    if( isAuthenticated()  &&  Authenticator.isValidAcct ( userName , acct)){
48      state.observeAcctCommand(acct);
49    }
50  }
51
52 }
```

Figure 5.11. *Evolution of FTP authentication, Configuration 2. The* ***AuthenticationMonitor*** *declarations of Figure 5.4 are assumed but not shown. When we make reference to any of these, they appear boxed.*

as a pattern of events. In Figure 5.12, lines 14-17 capture this authentication specification. Using the same primitive tracecuts specified for our original DEPs authentication implementation, and adding the `acct` tracecut (line 6), we are able to represent just the contexts of interest. The `isAuthenticated` tracecut includes semantic action blocks to further constrain the recognition of our pattern such that only usernames and passwords that correspond are permitted (line 15), only usernames and accounts that correspond are permitted (line 16), and that further occurrences of `acct` affect the current account information only if the account specified corresponds to the current user (line 17). The use of `rebind` on line 19 permits for additional occurrences of `acct` to be rebound to `a`. The advice declared on lines 20-23 implements the `ACCT-NEEDED` reply behaviour if the session is not authenticated under the requirements of configuration 1. The advice on lines 25-29 is a example of how to extract the account information for use by commands requiring it (such as those described by the `stor` primitive tracecut).

C2 In Figure 5.13, Configuration 2 is considered. Building from the existing authentication tracecut `isAuthenticated`, we first declare a primitive tracecut `Acct`. Unlike the `acct` tracecut used in the last example, this primitive tracecut includes a semantic action block that filters occurrences of the execution of `AcctCommand` to those that occur only after the session is authenticated, and only if the account information corresponds to the currently authenticated user. Lines 13-17 show this tracecut. In lines 19-21 the original `isAuthenticated` tracecut appears. However, we have opted to expose the username. It was used on line 15 to help validate the account occurrence. The most recent occurrence of account is maintained with a tracecut named `isAcct`, which we declare on lines 24-25. Again, `rebind` is used to permit repetitive binding of `a`. This tracecut would appear to capture our specification pattern faithfully.

```
1
2 public aspect AuthenticationMonitor_Config1 {
3   /* Observation pointcuts */
4
5   tracecut user(String usr): entry(user(usr));
6   tracecut pass(String pwd): entry(pass(pwd));
7   tracecut other(): entry(other());
8   tracecut acct(String act): entry(acct(act));
9   pointcut acct(String acct): execution(* AccountCommand.perform(String, Session)) && args(acct);
10  pointcut stor(String file): execution(* StoreCommand.perform(String, Session)) && args(file);
11  pointcut quit(): execution(* QuitCommand.perform(String, Session));
12  pointcut needAuthentication(Session s):  other() && !quit() && args(s);
13  pointcut needAcct(Session s):  stor(String arg) && args(s);
14
15 /* Event pattern for modified policy */
16 tracecut isAuthenticated(String acctName) {: String name, String pwd, String a :} ::=
17       user(name) pass(pwd) {: if(!Authenticator.isValid(name, pwd)){ fail; } :}
18       acct(acctName) {: if(!Authenticator.isValidAcct(name, acctName)){ fail;} :}
19       ( acct(rebind(a)) {: if(Authenticator.isValidAcct(name, a)){ acctName = a; }:} | other() )* $;
20
21   /* Behaviour modification */
22   void around(Session s): needAcct(s) && !history(isAuthenticated(String)) {
23      ControlConnection cc = s.getControlConnection();
24      control.send(new NeedAccountReply());
25   }
26
27   before(String account,Session session):
28     stor(String,session) &&
29     history(isAuthenticated(account)) {
30         Session.setAccount(account);
31   }
32
33   void around(Session s): needAuthentication(s) && !history(isAuthenticated(String)) {
34      ControlConnection cc = s.getControlConnection();
35      control.send(new NotLoggedInReply());
36   }
37
38   void around(Reply r):
39     cflow(pass(*)) &&
40     args(r) &&
41     call(* * ControlConnection.send(Reply)) &&
42     !history(isAuthenticated(String)) {
43         proceed(new LogInFailedReply());
44   }
45 }
46
```

Figure 5.12. *Full authentication with ACCT extension using DEPs, Configuration 1*

```
1
2 public aspect AuthenticationMonitor_Config2 {
3   /* Observation pointcuts */
4   tracecut user(String usr): entry(user(usr));
5   tracecut pass(String pwd): entry(pass(pwd));
6   tracecut other(): entry(other());
7   tracecut acct(String act): entry(acct(act));
8   pointcut acct(String acct): execution(* AccountCommand.perform(String, Session)) && args(acct);
9   pointcut stor(String file): execution(* StoreCommand.perform(String, Session)) && args(file);
10  pointcut quit(): execution(* QuitCommand.perform(String, Session));
11  pointcut needAuthentication(Session s):  other() && !quit() && args(s);
12  pointcut needAcct(Session s):  stor(String arg) && args(s);
13
14 /* Primitive tracecuts for filtering account messages so they are seen only when authenticated */
15 tracecut Acct(String account) {: String name :} ::= entry(acct(account))
16  && history(isAuthenticated(name)) {:
17           if(!Authenticator.isValidAcct(name, acct)){ fail; }
18   :};
19
20  /* Event pattern for original policy */
21  tracecut isAuthenticated(String name) {: String pwd :} ::=
22        user(name) pass(pwd) {: if(!Authenticator.isValid(name, pwd)){ fail; } :} other()* $;
23
24  /* Event pattern for most recent Acct policy */
25  tracecut isAcct(String acct) {: String a :} ::=
26        (user(String) | pass(String))* (Acct(rebind(a)) {: acct = a; :})+ $;
27
28
29   /* Behaviour modification */
30   void around(Session s): needAcct(s) && !history(isAuthenticated(String)) {
31      ControlConnection cc = s.getControlConnection();
32      control.send(new NeedAccountReply());
33   }
34
35   before(String account,Session session):
36     stor(String,session) &&
37     history(isAuthenticated(account)) {
38         session.setAccount(account);
39   }
40
41   void around(Session s): needAuthentication(s) && !history(isAuthenticated(String)) {
42      ControlConnection cc = s.getControlConnection();
43      control.send(new NotLoggedInReply());
44   }
45
46   void around(Reply r):
47     cflow(pass(*)) &&
48     args(r) &&
49     call(* * ControlConnection.send(Reply)) &&
50     !history(isAuthenticated(String)) {
51         proceed(new LogInFailedReply());
52   }
53 }
54
```

Figure 5.13. *Full authentication with ACCT extension using DEPs, Configuration 2*

5.3.3 Results

While various details of language syntax and tool support could be critiqued, we are interested in the more fundamental properties of the ability of each technique to achieve traceability, comprehensibility, and evolvability. Table 5.1 summarizes our comparison. An entry has been added for a putative FSM generation technique in which the developer must declare the states of the FSM explicitly, and the state transitions are specified as regular expressions. Such a technique corresponds to the approach of some of the related work we shall discuss in Chapter 6.

Manual instrumentation for the purposes of generating events necessarily requires scattering and tangling details in the base code, since the points where events are generated occur widely over the system. As a result of such scattering, an implementation must assume that all its parts will check and make use of the authentication state of the session in the appropriate manner. Each event potentially causing a transition in the finite state machine representation must be announced to the monitor; therefore, the base design must be manually instrumented to announce those events to the monitor. If any part performs this duty incorrectly, the user authentication protocol would be violated. Detecting and correcting the source of such an error would be (and was) difficult. Only the Java-extended version had this trouble.

Most of the approaches required manual implementation of the event parser, which

App	I	P	Tr	Un	Ev	Ex
Java	man	man	low	low	low	high
AspectJ	gen	man	medium	low	medium	high
EAOPTool	gen	man	medium	low	medium	high
FSmedium gen.	gen	gen	medium	medium	medium-high	low
DEPs	gen	gen	high	high	high	medium

Table 5.1. *Comparison of different extension approaches. Table entries indicate man(ual) vs. gen(erated) or L(ow), M(edium), or H(igh); categories are: means of **I**nstrumentation; means of creating **P**arser; and resulting **Tr**aceability, **Un**derstandability, **Ev**olvability, and **Ex**pressibility.*

necessarily obfuscates the patterns of interest; however, manual implementation permits the greatest expressibility since a Turing-equivalent language is available. In practice, this expressibility is more a burden than a boon. For FSM-based generators, the language that can be recognized is poorly expressive. Furthermore, the specification of the patterns is difficult as specific states must be identified in conjunction with the events that cause transitions. Parser generators, on which our proof-of-concept tool is based, do not require that states be explicitly identified, but can generate them based on the interacting patterns of interest (i.e., production rules) as described by the developer.

The use of tracecuts improves comprehensibility over the use of context variables such as the `isAuthenticated` field in the AspectJ version or the explicit states that had to be identified in the finite state machine represented in Figure 5.2. The developer can see through the tracecut declarations just how a history is to be satisfied. Context variables on the other hand are disconnected from their intent. Although their name may offer an expectation as to their purpose, confirming, refuting, or modifying that purpose requires delving through potentially complex implementations. This increases the probability that a developer will effect changes to a system when their understanding of it is insufficient. On the other hand, a tracecut—being a more direct implementation of a concept—reduces the likelihood of such an error.

The evolvability of an approach is limited when it causes scattering and tangling of event instrumentation or obfuscation of protocols in implementing the event parser. Thus, the evolvability of the Java approach is poorest, and that of AspectJ and the EAOPTool is intermediate. The FSM-based generator and DEP approaches generate the parser from a high-level specification that is simpler to modify. The modifiability of the FSM-based approach is reduced because it is relatively difficult to define a new FSM manually should the requirements change. Our own experience with defining finite state machines from informal specifications, as well as the experience of others [97], supports this contention. From Figures 5.9 and 5.10 it is clear that

a full consideration of the FSM implementation was required to evolve our authentication policy to include `ACCT` configuration 1. Every modular unit in this example was touched in some matter, either by altering or adding adding state transitions, or consideration of the effect of existing transitions. The boxed areas in these figures mark each change or addition in that source listing to highlight the demanding task that such FSM implementation requires. These changes were extensive and tedious to conduct.

We were able to represent various evolutions of our regular expressions for user authentication directly with the use of declarative event patterns as provided by our proof-of-concept tool—no translation to more complicated representations or models was required. Authentication becomes a simple statement on patterns of events, resulting in simplification and localization of the state specification as compared to the other solutions.

5.4 Summary

This chapter has seen the application of declarative event patterns in two representative cases: path-specific customization (5.2) and protocol/policy enforcement (5.3). In conducting these case studies, we sought to provide qualitative evidence supporting the claims that DEPs provide improved expressiveness and improved evolvability of context-senstive crosscutting concerns when compared against current approaches. Our results indicate that this is indeed the case.

The evidence we collected in these studies points in favour of our claims that declarative event patterns do help to capture context-sensitive crosscutting concerns in a manner that is more understandable, traceable and evolvable when compared against other approaches. In particular, where object-oriented and aspect-oriented approaches failed to cleanly capture our context-sensitive crosscutting concerns, DEPs improved the overall implementation of these concerns along all three dimensions.

They were seen to address not only the scattering and tangling of crosscutting concerns, but also the difficult task of recognizing context.

Chapter 6

Related Work

In this chapter, we discuss work related to declarative event patterns. We categorize related work into the following groups: foundations of aspect-oriented software development (Section 6.1), monitoring, verification and specification approaches for exploiting context-sensitivity of programs (Section 6.2), and programming approaches leveraging the communicative context of a program's execution (Section 6.3).

6.1 AOSD foundations

Aspect-oriented software development is an umbrella term encompassing a spectrum of research aimed at improving software development; the field derives from metaprogramming, genericity, generators, reflection and other *post-object* approaches [21, 30]. The commonality throughout these works is their consideration of new and more expressive modularization and decomposition mechanisms. Declarative event patterns build on a significant number of contributions from this area of research.

6.1.1 Founding Approaches

Aspect-oriented programming (AOP) promotes the separation and modularization of the crosscutting concerns in a system [30, 51]. By separating a system into multiple distinct concerns, the goal of AOP is more evolvable, comprehensible, adaptable, customizable, and reusable systems. To form a functioning system, the separated

concerns of the system must still interact. We have looked at one view of aspect-oriented programming throughout this thesis. However, there are others worth noting.

Aspect-oriented programming (AOP) is tied closely to the early work into aspect-oriented programming done at Xerox PARC. Of this flavour of AOP, AspectJ is the most prominent and well supported realization. AspectJ employs an asymmetric view of composition, distinguishing between base concerns (represented in standard Java code) and crosscutting concerns (represented in a modular unit known as an aspect). Points that have meaning in the execution of the base code—*join points*—are associated with crosscutting code by way of expressions called *pointcuts* and method-like declarations called *advice*. The process of combining aspects with base concerns is called weaving, and may occur at compile time, load time, or dynamically. Declarative event patterns (DEPs) are an extension of this model and so inherit its features and disadvantages. DEPs gain the benefit of a rich set of join points. Events used in DEPs for establishing context are specified in terms of the combination of AspectJ join points and before and after advice. As a result all current and future join points supported by AspectJ are automatically supported by DEPs. However, AspectJ operates in a concrete referential context, or *shared world view*, and uses an asymmetric decomposition paradigm. Most notably, it lacks a general means for relating events execution context.

Subject-oriented programming (SOP) [64, 39] is a compositional approach to developing large scale software systems. With SOP, the essential qualities of an object depend on the observer's subjective view of the world. SOP research has shown that object-oriented decomposition alone does not allow for more than a single perspective on the intrinsic properties of an object and its classification. In this approach developers decompose a system into multiple separated subjective decompositions—*subjects*. These subjects are typical object-oriented decompositions, but only focus on those parts of the system important to the concern being addressed. At composition time, these separate subjects are combined using composition rules. These rules

permit common entities in different subjects to be merged, overridden or otherwise combined until a concrete system results. Multi-dimensional separation of concerns (MDSOC) [65, 84], the successor of SOP, argues that existing means of software decomposition and composition suffer from supporting only a single dominant dimension of separation at a time; because of this, they fail to capture overlapping concerns along alternate dimensions of decomposition. MDSOC has been partially realized in the Hyper/J tool. Hyper/J provides two constructs to achieve its goals: hyperslices and hypermodules. A hyperslice is a set of behavioural units (Java methods and classes) drawn from any existing (and compiled) system which together represent a concern. Hypermodules are also introduced as a unit of composition of hyperslices. Like the subject-oriented paradigm, these sets of hyperslices are composed using composition rules to merge or replace common concepts in each hyperslice. The result of this composition is another unified hyperslice. Both SOP, and MDSOC have provided foundation to the AOP notion of independent views or perspectives of a system. Symmetric composition and subjectivity are these works primary contributions.

Composition filters, one of the earliest approaches that falls under AOSD, provides an object model where the messages sent and received may be intercepted, acted on, redirected, forwarded, or delegated at runtime. Filters may be applied to classes in order to alter or extend the set of messages it may receive. Filters may represent orthogonal concerns; a single filter may be applied to more than one class. Composition filters operate on individual messages, and hence do not provide support for utilizing communicative context.

Adaptive programming (AP) [56, 55, 52] is a technique for under-specifying dependencies on the static structure of an object-oriented system. It is a practical approach that incorporates the *Law of Demeter* (LoD), which states that modules should "only talk to their friends", and "traversal specifications", which aim to bridge connections between structurally separated modules. The key benefit of this "structure-shy" approach is that it leads to better information hiding; explicit dependence on the static

structural context of the system is limited. At composition time, the specified traversals augment the concrete code of the system to automatically provide the necessary intermediate dependencies. The join point model involves the nodes and edges of the class hierarchy and collaboration diagrams. The idea of communicative context is similar in principle to "structure-shy" programming. With AP behaviour is defined relative to static structural patterns between classes; declarative event patterns specify behaviour relative to dynamic patterns sought in the execution. Both approaches share the notion of context-insensitivity (under-specification).

6.1.2 Nature and Claims

The nature of AOP itself has been explored. Sullivan *et al.* [81] have looked at the relativity of crosscutting concerns, given a set of mechanisms to decompose (and recompose) a system. They bring into question the limits to which any one language can succeed in expressing a modular construction of a design structure and re-cast aspects as just the latest in a long history of language mechanisms invented to account for the limitations of existing languages to map design level decisions to program-level modularity. Declarative event patterns are another mechanism in this line.

Filman *et al.* [33] advocate the view that AOP is the quantification of events that occur in the execution of a program, combined with a modules obliviousness to such quantification. They take obliviousness in this context to mean that modules being composed are not aware of potential modifications that could result. There is no indication of dependence between modules being composed.

The aspectual decomposition is not without its skeptics though. By now there has been a fair number of studies measuring the effectiveness of AOP to its claims [61, 90, 60, 12, 19]. Though nothing is conclusive, evidence is mounting. In [90] Walker *et al.* explore the benefits and drawbacks that AOP places on developers when reasoning about and changing software. Their results indicated that the nature of the separation of code has much to do with the benefits of the separation. When the interface

between the seperated code and the base code was small ("*narrow*") the participants didn't have to think much to understand the separated code. The "*wider*" this interface became, the more they had to consult the base code and reason about the effects of the aspect in the base code. Specifying separated code in a more intentional, declarative fashion is important to easing the developers cognitive burden. When the relationships between points in a wide interface is evident from the aspect specification, one is able to reason more locally. Declarative event patterns are both declarative and aim to reveal the intended relationship between events.

6.2 Monitoring and Verification

Various techniques make use of event traces or historical references for purposes other than implementation. For example, in formal specification techniques, such as trace assertion [13]; or run-time verification techniques, such as intrusion detection [87]. Colcombet and Fradet [20] describe a theory behind the enforcement of trace properties for the sake of detecting security violations; this work is restricted to regular languages and does not address issues of comprehensibility. Reiss and Renieris have described a technique for compressing voluminous execution traces as they are collected, through an encoding based on CFGs and construction of automata [71]. Declarative event patterns use similar techniques but specifically for the purposes of high-level implementation.

The work of De Pauw *et al.* [67] attempts to automatically discover emergent patterns in executions, rather than specify behaviour that should execute when expected patterns occur.

Approaches that draw on model checking and temporal logic have been employed to describe execution properties [88, 94, 4]. These approaches are limited to the realm of design and model checking, require abstraction of states, and have no direct means of implementation. Åberg *et al.* [1] have used a branching logic to specify points in

which instrumentation should be included in an operating system project. This work was limited in that extent that advice could be applied, and did not provide a means to bring context forward. We have found that the use of temporal logics, while formal, are not always not always easy to comprehend or evolve. Declarative event patterns use context-free grammars in the hope of alleviating this.

Filman, Havelund, and their colleagues have used largely manual instrumentation and event parsing techniques, particularly for verification (e.g., [35, 34]). Their work is primarly interested in runtime monitoring of safety critical systems. In Havelund and Roşu [40, 41], a linear temporal logic is presented as a language for stating safety properties an used in off-line analysis of the execution of the system. They describe a system in which temporal property monitors are constructed and used to analyse a complete execution trace. DEPS differ from this work on a number of fronts. They use a linear temporal logic to expression properties. They are primarily interested in observing that properties hold; they do not extend this as DEPs does to influencing execution should a property be observed. In all these approaches the emphasis in on what can be expressed, not the ease with which they can be expressed. Declarative event patterns focus on the software engineering benefits to be gained by a proper consideration.

6.3 Communicative context

Larochelle *et al.* [53] propose a solution space where by the "visibility" of join points is limited, refined and encapsulated. Declarative event patterns are one such solution in this space, refining the selection of a join point based on the communicative context of an executing program.

Roles are an approach for noting objects' participation in collaborations [38, 86]. While acknowledging that objects may act in different ways depending on the current collaboration, the collaboration itself is not modular in implementation. Design pat-

terns, for example, provide patterns of interaction between abstract objects playing certain roles [36]. While providing a design for the collaboration, it is often difficult to see what patterns an object participates in based on the implementation alone. Declarative event patterns make important interactions significant and intentional.

The selection of behaviour based on runtime properties, polymorphism, is a facet of many languages. Dynamic dispatch, double dispatch [45], multiple dispatch (polymorphic on all arguments), and more generally predicate dispatch [31] are some such approaches. Declarative event patterns permit the selection of behaviour based on the execution context itself. Declarative event patterns leverage this implicitness in order to monitor communicative context, and are a partial realization of contextual dispatch [89].

Implicit invocation [96, 37] inverts the calling dependency of normal object collaboration. Explicit implicit invocation [96] would have the original callee *register* with the original caller, to be invoked at another time (publish–subscribe). On the other-hand, Aspect-oriented approaches permit implicit implicit invocation. The callee no longer is dependent on the caller; the registration is made implicit. Declarative event patterns build on this implicit registration in order to observe contextual events in the execution and affect semantic change on the program. A contextual specification expressed with DEPs is a kind of declarative registration; those points in the base program that correspond to primitive events need not register.

6.3.1 Protocols and Traces

The ability to have dynamically scoped variables has existed from very early on; it featured in earlier LISPs for example (albeit unintentionally). Within the extent of a call, a variable may bind to the most closely scoped referent. This flexibility is unstructured however, and is limited to the control flow under which the variable is referenced.

Coady and Kiczales [19] have evaluating the potential for AOP to improve the mod-

ularity of OS kernel code. In this work they note a number of crosscutting concerns that are better represented as aspects. In their work, they use a `cflow`-like construct. `cflow` is a limited form of communicative context; a join point may be quantified not only by the type of join point itself, but also by whether it is occurring within ther dynamic extent, the control flow, of another specified join point [50]. Declarative event patterns loosen this restriction, and permit the communicative context to extend indefinitely into the past execution. Of these, path-specific customization, such as our Columba example, are prominent.

Proebsting and Zorn [69] have explored what they call "tangible program histories" to represent previous program state that is not explicitly recorded by the programmer. Examples of their program histories mechanisms include auto-accumulators for a variable, and counters that represent the number of times a given loop has iterated. This work appears to have been abandoned.

Douence *et al.* [26] have advocated an event-based approach to aspect-oriented programming, realized in the EAOPTool [27]. The EAOP work aims to generalize the notion of aspect and crosscuts. Events that occur throughout the execution of the system are emitted to a co-routine monitor. The monitor interprets the events in a generally imperative manner, reminiscent of the main event loop found in graphical interfaces. The aspect monitor is free to interpret the stream of events in whatever way the application demands. Their approach is supported by a rudimentary tool. Fundamentally, their approach has shown that using events as the basis for aspects is feasible. Later, Douence *et al.* established a theoretical framework for "stateful aspects" [24, 25], which can be used for analysis of event patterns. Unlike declarative event patterns, this work does not focus on the software engineering principles; writing monitoring aspects with their tool is difficult and prone to error [93].

A number of techniques force the developer to explicitly identify states and state transitions. Type adaptation [97] allows modules to be composed via stateful translation protocols specified as finite state machines (FSM); however, the work has not

been extended beyond pairs of modules. State abstraction has been promoted as a mechanism for specifying behaviour in modular software development [43]. Butkevich *et al.* [15] use explicit state-and-transition representations of FSMs to aid in debugging object protocols. JAsCo [82], a tool that combines AOP and components, claims to provide stateful aspects as described by Douence *et al.* [26]; however, the FSM states and transitions in a protocol must be explicitly described. FSMs lack the expressibility needed to recognize properly nested structures in event patterns. DEPs allow states and state transitions to be defined implicitly via trace specifications. In some situations, states and state transitions are readily available, but in typical development settings, this is not the case. Yellin and Strom [97] agree, indicating that it is difficult to specify these state machines.

Serini and de Moor [78] consider the optimization of regular patterns of join points on the call stack. While their emphasis is on compiler optimization of such patterns, their work is important for eventual optimizations of declarative event patterns. This work is limited to the use of regular patterns and as of writing remains to be incorporated into existing technologies, such as the `abc` AspectJ compiler [5]. The members of this project are actively exploring compiler optimizations for AspectJ and other languages; a large part of their effort is in the reduction of runtime overhead induced by the dynamic features of the language, such as `cflow` [29].

De Moor *et al.* [28, 79] have been exploring the use of path logic programming for performing incremental optimizing transformations on programs. The paths described by this Prolog-variant are regular patterns over potential execution paths. While their patterns are less expressive when compared with DEPs, they can be efficiently recognized. The URD tool has been designed in such a way that "you don't pay for what you don't use". That is, a regular pattern expressed as DEP results in a contextual monitor without the baggage required (parsing stack) required of a context-free pattern.

As early as 1989, Van Roy [73] noted the usefulness of separating tangled concerns

along an execution path. He proposes a preprocessor which generalizes Prologs definite clause grammar (DCG) notation to allow the separation of multiple accumulators to act on a single path. This approach, while pre-dating aspects, is an example of the separation of tangled path concerns. Definite clause grammars allow the specification of a class of attributed unification grammars with semantic actions. These grammars are strictly more powerful than context-free grammars. When programming with DCGs it is often useful to carry more than one semantically distinct accumulator along a clause. Without Van Roy's extension, each accumulator requires the use of two additional arguments, and these arguments must be chained together. The result is the introduction of many arbitrary variables, and, as he points out, the chance of introducing errors. Furthermore, he notes that modifying or extending this code, for example to add another accumulator, is tedious. His solution extends the DCG notation to allow for an unlimited number of named accumulators, and handles all the tedium of parameter passing. Each accumulator requires a single Prolog fact as its declaration, much like a named advice. The bulk of the program source does not depend on the number of accumulators, so maintaining and extending it is simplified. In essence, the extension allows for each accumulator to be specified as a separate perspective. Our implementation of DEPs can be thought of as a kind of breadth-first traversal in this sense. Unlike DCGs, where the control flow is dictated by search, DEPs observe the execution of arbitrary object-oriented flow of control. Similarly, in this approach additional accumulators are "woven" along a dominant path, providing asymmetric composition.

Lopes *et al.* [58] have proposed *naturalistic programming* as a general direction for the post-AOP era of programming languages. Naturalistic languages incorporate features common to natural spoken languages such as anaphoric relations. Anaphors allow us as humans to reference elements in discourse. Declarative event patterns provide a kind of anaphoric reference to past events in the execution history.

As an extension of the work on communication history, declarative event patterns

(DEPs) borrows a great deal from implicit context. Declarative event patterns focus on the communicative context part of implicit context, and aim to be a more practical realization of the idea of communication history. A fundamental difference between declarative event patterns and implicit context is that declarative event patterns, built apon AspectJ, inherits a shared world view and assymetric composition model. Implicit context shuns both of these, preferring the modules with local subjective contexts, that can be composed symmetrically.

6.4 Summary

In this chapter we have reviewed existing and ongoing research related to declarative event patterns. In particular, works related in principle to declarative event patterns, and works sharing similar goals to declarative event patterns have been described and compared.

Chapter 7

Discussion

There are a number of avenues for consideration that are beyond the scope of this work, but should not be left unmentioned. In this chapter we consider the particular implications and limitations of our concrete approach to declarative event patterns, and outline a plan for future exploration.

7.1 Expressivity

Choice of Contextual Anchors We found that it is often the case that more than one pattern was possible for describing adequate communicative context. While some interactions require more constraining contexts, we found that there is flexibility in terms of join point locus along the communication path. In many cases, there was often a choice whether to distinguish between the entry and exit events surrounding a call or execution join points. In a practical sense we found that the distinction only matters if context is to be exposed; that is, whether the exposed context originates at the call site or the execution body. From the perspective of modular dependence, the choice of call or execution join point is also significant. One should avoid advising over too many modules.

Reuse and evolution The choice of contextual anchors is important in terms of reuse. Evolution of a system may result in important events disappearing, resulting

in a failure to capture suitable contexts. In our examples we have tried to select the most promising and general facets from the concrete context in which our DEPS are being applied.

Choice of Pattern Language Different languages are appropriate for different needs. DEPs address sequential relationships over the events in the execution path of a system. In general, event monitors for different relationships are possible and likely. Not all relationships are sequentially determinable. For example, enforcing structural constraints on the elements in a figure may require a more general constraint mechanism. The application of neural nets and expert systems to these monitors is an exciting possibility.

Why Context-free Patterns? In determining the most appropriate language for our contextual patterns we considered many different formalisms. Conjunctive grammars [62], Regular tree grammars (XPath, xml related), past-directed linear temporal logic, regular expressions as well as a more expressive form of the communication history API were explored. We settled on context-free grammars because they would be familiar to most computer scientists and permitted the expression of nested structure. Temporal logics were cryptic and regular tree languages awkward for underspecification of both self-embedding and iteration. Conjunctive grammars on the other hand offered an arguable more suitable formalism, but at the expense of more complex runtime.

The absence of self-embedding patterns in our case studies is interesting. Context-free protocols do exist, yet we found that in most cases, communicative context was expressible using regular patterns. It remains to be seen if there is a tendency for developers to reason about path context in this manner.

Is it Context-Free? In fact, the patterns over the execution trace of a program that we can recognize with DEPs are not quite context-free. While the underly-

ing mechanisms for recognizing events accepts context-free grammars, and generates parsers that strictly can only recognize context-free languages, with the use of primitive tracecuts we can selectively expose or conceal events in the stream of execution. Only those events mentioned in the pattern are considered—a closed universe of events. The loosening of the immediateness of context-free dominance can help, but also forces awkwardness in other patterns that require immediacy. In particular, in the FTP example (see Chapter 5) authentication required the inclusion of `OTHER` events, such that we could insist on `USER` events being immediately followed by `PASS` events. Had we not included other events, our parsers would implicitly skip over any event that may potentially slip between `USER` and `PASS`; in this case that would be undesirable.

In the case of the ACCT evolution of the FTP authentication protocol, we were able to achieve local lexical extents for particular context in our grammar. Whether or not our parser observed the ACCT event depended on the context already established. The result is fundamentally more expressive than could be achieved had each event in the complete stream of events required consideration, on par with the extended domain of locality gained with lexicalized grammars [16].

Why AspectJ? Rather than provide declarative event patterns atop AspectJ, we could have chosen a different base or to start from scratch. We chose to extend AspectJ both to take advantage of its existing features and to simplify comparisons. However, the combination of AspectJ and DEPs is imperfect. DEPs operate on events while AspectJ operates on join points. What constitutes a join point depends on the join point model in sway for a given language. In AspectJ, method execution is considered a join point, but it is not an event in the sense described above since it cannot be instantaneous. It is the combination of advice and pointcut that determines when an "event" really happens. This difference can lead to subtle problems; however, this does not imply a shortcoming in the declarative event patterns approach. An

industrial-strength realization of DEPs would need to reconcile the differences more cleanly. AspectJ is arguably the most well-known, and well-supported AOP realization available; a large base of users from both research and industrial are familiar with its syntax and capabilities. As a language, AspectJ already provides sufficiently low-level building blocks (namely, pointcuts and advice) on which to support the higher-level DEP abstractions.

Communicative Events Specification of interest in particular event equivalence classes has another advantage: backwards compatibility in the face of extensions to a language's join point model. Some primitive pointcuts (such as `cflow`) select all the join points of any kind that occur within a lexical or dynamic extent. If the definition of what constitutes a join point is extended, advising one of these pointcuts will cause different program behaviour depending on what version of a language is being used. In a fully event-based approach, we would have only specific events cause tokens to be sent to a parser. Therefore, extending the range of what event classes could be specified would not affect existing source code regardless of what version of a language it were compiled under.

Primitive tracecuts encompass join points specified by standard AspectJ pointcuts. Pointcuts do not necessarily specify disjoint sets of join points; events may equally fall under more than one pointcut, and so one join point may underlay multiple communicative events. The current implementation of DEPs would observe and consume multiple events when primitive tracecuts overlap. The order in which these events are seen, from the standpoint of our prototype implementation, is determined by the appearance of the primitive tracecut declaration in the source file. We have conducted initial investigation into the use of general LR recognition coupled with the Schrödingers token approach [9] as a means of allowing a join point to simultaneously be a member of multiple terminal classes. Other approaches for resolving lexical ambiguity look promising [14].

7.2 Challenges

Dynamic loading issues The dynamic introduction of code could pose difficulties for our technique in some situations. If the loaded code contained DEPs that required access to details of the communicative context that were not observed before loading, these DEPs could not be evaluated conservatively. This weakness would be equally present in a system not using DEPs, as the existing code would need to have kept track of state that might only be of interest to the dynamically introduced code. Further research is needed to address this issue, with or without DEPs. Storing events in sequence over a short window may provide some of the missing context, but in general, as we have argued, communicative context involves events that may occur at any point in execution, and so no guarantee could be met by such an approach.

Concurrency DEPs inherit from AspectJ a single-threaded model of patterns. While it is possible to explicitly match patterns across multiple threads, in practice, we have very rarely used multi-threaded queries. Regardless, the tool implementation can be extended to deal with certain limited cases of multi-threading. Synchronization must still be considered when using declarative event patterns. Race conditions between updates to automaton state and accesses to that state could occur unless the automaton methods were synchronized. Total synchronization would be straightforward to implement; more efficient synchronization is likely possible, but remains non-trivial future work.

Similarly, communication between distributed objects is not immediately handled by DEPs. Communication events occurring on a remote node must be handled appropriately.

7.3 Other considerations

Tool limitations Our choice to use reduced-parse-stack activity parsers [8, 7, 75, 74, 76, 77] is both a benefit and a drawback. Because much of the context is encoded in the parser itself, the complexity of our patterns are limited. An involved grammar (such as those required for the syntax of Java) result in excessively large parsing automata. However, our implementation permits the use of any context-free grammar, and so it is possible to express ambiguous languages of events that call for a generalized parsing approach. In a generalized approach, as we have seen, multiple processing elements are active concurrently—each exploring an alternate interpretation of the observed stream of events. Each of these processing elements maintains a stack to hold its place in context. Reducing the size and number of these stacks is an important consideration for any generalized parsing approach.

Another consideration in regards to multiple processing elements is in that of memory leaks. It is possible that a processing element might wait indefinitely for an event to occur. While it waits, the parsing stack, and references contained in its binding environment are also held indefinitely; they cannot be reclaimed by the garbage collector. We plan to address this issue in future revisions of our tool.

We were not able to leverage existing GLR tools such as Bison [23]. Dynamically recognizing events in a programs execution requires that only a minimum amount of event information be retained. Existing GLR parsers assume finite file based input, and leverage this complete record of the input in constructing. This is not a feasible option for tracecuts.

Formal Specification versus ease of use Formal specification is difficult to use in practice. DEPs provide a compromise between specifying and implementing protocols by combining the specification and implementation steps. This narrowing of the gap between implementation and specification promotes rapid exploration of alternate context specifications, and permits a more streamlined development process.

Traces vs. State machines The finite state machine (FSM) representation of the user authentication protocol caused difficulties because it needed to be derived from a few informal descriptions within the FTP specification. One might argue that the FTP specification should have been more formal in the first place, and then this difficulty would have been mitigated. However, someone would have still needed to have generated the FSM in order for it to be present in the specification; this would remain a difficult task. Yellin and Strom have reported similar difficulties in generating state machine representations of translation protocols for type adaptation [97]. The informal specification of FTP describes its key properties, and that was considered sufficient by its creators across its many years of development. Similarly, one could ask whether a temporal or modal logic would be more appropriate for the expression of event patterns. Our initial attempts at providing an event pattern language were based in temporal logics and ended with unsatisfactory results. In some situations, temporal logics may be more intentional than DEPs; however, neither a CFG- nor a logic-based approach is likely to be ideal in all circumstances for all purposes. State machines constitute concise, complete descriptions of behaviour for protocols. A state machine may be defined implicitly via a description of some set of paths through it. If this path description underspecifies the state machine, this can indicate some details are flexible; such flexibility is preferable to selecting and enforcing arbitrary constraints that may need to change at a later time. Declarative event patterns permit such flexibility by specifying only the paths of importance.

7.4 Future work

Parameterization Adding the ability to define parameterized tracecuts would permit the encoding and reuse of commonly-occurring patterns. These generic patterns would effectively define new operators in terms of structural and temporal properties of event traces. The `cflow` pointcut is a hard-coded example of what a generic

pattern could define.

Negation DEPs lack a means for specifying the negation of events. The semantics of negation in terms of tracecuts could however take on different meanings. For example the negation of a primitive tracecut `!user()` could either mean *any* event in the execution context except those specified by `user()`, or it could be closed to only other events mentioned in the pattern in which the negation appears. Similar issues could be raised for ordered tracecuts as well.

7.4.1 Optimizations

Up until now, we have presented declarative event patterns in terms of their characteristics for improving software evolution. This is the primary aim of this thesis, but the adoption of such approaches inevitably is tied to practical application. The URD tool offers a purely dynamic approach to the observation of event patterns, and in the validation of our approach, we did not feel any adverse overhead of the additional runtime mechanisms. In contemplating the optimization of declarative event pattern approaches, the leveraging information obtained statically from the program is a prime candidate. The overhead induced through runtime monitoring should be limited as much as possible. How much static information can help depends on both the program and the patterns involved. A number of elementary optimizations have been considered. In order to fully consider the possible execution paths that may match against a DEP, analysis must be conducted post-weave; that is, all potential advice must be interwoven into the base code. If this was not the case, the analyzed paths would not be reliable.

7.4.1.1 Grammar Reduction

One optimization would act on the pattern itself. An analysis of the source code would determine, when possible, if an event mentioned in a DEP was even possible

in the base execution. Tracecuts that mention such absent events would be removed from the grammar. If the simplified grammar was still realizable, further optimization would commence.

7.4.1.2 Regular approximation

By determining a regular approximation of the context-free grammar, and using practical approaches for analysing regular paths in execution, we can arrive at a superset of paths that contain all the paths conforming to the grammar. This analysis would further limit the paths of consideration and separate the grammar into regular and Dyck parts, the regular parts being the approximation, and the Dyck part forming the basis of an optimized runtime monitor. Dyck language recognition (i.e., cflow) can be optimized, and can sometimes be reduced to a set of counters, or better still, optimized away completely with further call analysis [6].

7.4.2 Implicit Context & Registration

Declarative event patterns have not yet been incorporated with IConJ, a tool for implicit context. Merging DEPs into the implicit context paradigm would require two initial changes. Currently DEPs assume a globally shared referential context. Implicit modules have subjective contexts. Hinted in the quote at the beginning of Chapter 1, communicative context is *interstitial*, that is, it occurs in between modules. Aspects augmented with declarative event patterns allow one to specify context in code. As a result, the implementation is locally easier to understand, trace, and evolve. However, when specifying context, by the very nature of the problem, one must consider the structure of the system in a manner that is non-local. In order to specify contextual markers, one must gain a non-local (but not necessarily fully global) understanding. IConJ supports subjective renaming of types, and specification of contextual dispatch often considers the *inbound* and *outbound* distinctly. Secondly, a new syntax would be required to meld the two approaches together.

Implicit context aims to permit unanticipated evolution and reuse, not by invasively modifying modules, but instead, by altering the underlying context of their operation. This separation of operating context opens doors for a more dynamic consideration of implicit reference systems for programming languages. Registration is a term used in [80], and involves the continual interpretation, or registration, of objects in subjective contexts. It provides a fluidity such that as context changes, so too may the referents of the system. Implicit context teases away this registration of context in the form of boundary maps. Communicative context enhances maps with a form of dynamic rebinding. Future work in further separating the reference system from the language is a promising avenue of research. Such an approach may employ constraint programming discourse representation theory, and other areas of computational semantics.

Chapter 8

Conclusions

In software systems comprised of many modular parts, the flow of communication between the modules shapes the behaviour of the system as a whole. In the absence of some automatic means of interpreting (or re-interpreting) events in the *execution context* as they transpire at runtime, today's software developers make use of *protocols*. Protocols script interactions and exist to provide a clear and indisputable interpretation of the semantics of a communication act given what has transpired up until that point. They can be explicitly acknowledged and adhered to, as is often the case with application level protocols such as the File Transfer Protocol (FTP [68]), or implicitly assumed as is often the case with the interactions between individual classes or interfaces found in object-oriented software systems. Whether explicit or implicit, adherence to a protocol provides a kind of pre-defined semantic interpretation; if the protocol is followed, a particular context may be assumed.

A problem with realizing protocols in software systems is that each participating module becomes responsible for playing out its own part in the conversation; consequentially, partial knowledge of the larger communication is necessarily split up and dispersed amongst each participating module; this is termed *scattering*. This scattering of knowledge leads to the "disembodiment" of the protocol as a whole. The protocol also becomes tangled amongst the details that belong inside other modules; this is termed *tangling*. Scattering and tangling together make it difficult to locate, understand, and change the protocol, the affected modules, and the system

overall [84].

We hypothesized that an approach that permits the recognition of program execution contexts to be specified and implemented in a localized way, separate from the details of the participating modules, might aid the evolvability, traceability and comprehensibility of context-sensitive crosscutting concerns, at least when compared against existing popular approaches. *Declarative event patterns* (DEPs) are a means to implement context-establishing protocols between modules while retaining the ability to locate the parts of a protocol realization (traceability), understand the implementation of a protocol (comprehensibility), and more easily alter the protocol and its realization (evolvability). Contained within a modular unit of their own, they describe sequences of events in the execution of a system; these sequences represent execution contexts of interest—properties of the context that must hold in order to enable the specific semantic interpretation of a given communication act.

We validated our hypothesis by first constructing URD, a prototype tool for declarative event patterns. Using URD we were able to validate our claims of improved traceability, evolvablility and comprehension by conducting a number of case studies—two of which we reported on in this thesis. What we found was that declarative event patterns aided the developer along our goal dimensions when compared against Java, AspectJ and other approaches. Using DEPs, modular implementations of context-sensitive concerns were possible and shown to be less difficult to evolve, understand and trace back to original specifications. The tedious and error-prone processes required for recognizing context in Java and AspectJ were shown to be absent in DEP implementations.

This work makes the following contributions.

1. We proposed and implemented a practical approach for the modular implementation of context-sensitive crosscutting concerns [93] based on aspect-oriented programming.

2. We pointed out and demonstrated that important software engineering proper-

ties of a program's implementation (evolvablility, comprehensibility, and traceability) are achievable using DEPs.

Bibliography

[1] R. A. Åberg, J. L. Lawall, M. Südholt, and G. Muller. Evolving an OS kernel using temporal logic and aspect-oriented programming. In Y. Coady, E. Eide, and D. H. Lorenz, editors, *The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Mar. 2003.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] M. Akşit, editor. *Proc. 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD-2003)*. ACM Press, Mar. 2003.

[4] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 187–209. Springer-Verlag, 2001.

[5] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[6] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising aspectj. *SIGPLAN Not.*, 40(6):117–128, 2005.

[7] J. Aycock, N. Horspool, J. Janousek, and B. Melichar. Even faster generalized LR parsing. *Acta Informatica*, 37(9):633–651, 2001.

[8] J. Aycock and R. N. Horspool. Faster generalized LR parsing. In *International Conference on Compiler Construction (CC 1999)*, volume 1575 of *Lecture Notes in Computer Science (LNCS)*, pages 32–46, Amsterdam, March 1999. Springer.

[9] J. Aycock and R. N. Horspool. Schrödinger's token. *Software—Practice & Experience*, 31(8):803–814, 10 July 2001.

[10] C. Babbage. Charles babbage quotes. http://www.saidwhat.co.uk/quotes/c/charles_babbage_1997.php.

[11] E. Baniassad and S. Clarke. Finding aspects in requirements with Theme/Doc. In B. Tekinerdoğan, P. Clements, A. Moreira, and J. Araújo, editors, *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, pages 15–22, Mar. 2004.

[12] E. Baniassad, G. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: An inquisitive study. In Kiczales [48], pages 120–126.

[13] W. Bartussek and D. L. Parnas. Using assertions about traces to write abstract specifications for software modules. In *Information Systems Methodology*, volume 65 of *Lecture Notes in Computer Science*, pages 211–236, 1978.

[14] A. Begel and S. L. Graham. Language analysis and tools for ambiguous input streams. *Electr. Notes Theor. Comput. Sci.*, 110:75–96, 2004.

[15] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and tool support for debugging object protocols. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 50–59, New York, NY, USA, 2000. ACM Press.

[16] J. Carroll, N. Nicolov, O. Shaumyan, M. Smets, and D. Weir. Parsing with an extended domain of locality. In *Proceedings of the ninth conference on European chapter of the Association for Computational Linguistics*, pages 217–224, Morristown, NJ, USA, 1999. Association for Computational Linguistics.

[17] M. Chu-Carroll. Software configuration management as a mechanism for multidimensional separation of concerns. In P. Tarr, A. Finkelstein, W. Harrison, B. Nuseibeh, H. Ossher, and D. Perry, editors, *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, June 2000.

[18] S. Clarke and R. J. Walker. Generic aspect-oriented design with Theme/UML. In Filman et al. [32], pages 425–458.

[19] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 88–98. ACM Press, 2001.

[20] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–66, New York, NY, USA, 2000. ACM Press.

[21] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[22] E. W. Dijkstra. *A Discipline of Programming*. Series in Automatic Computation. Prentice Hall, Englewood Cliffs, United States, 1976.

[23] C. Donnelly and R. M. Stallman. BISON—the YACC-compatible parser generator. Technical report, 1988.

[24] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 173–188, London, UK, 2002. Springer-Verlag.

[25] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM Press.

[26] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, 2001. (3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns; REFLECTION 2001; Kyoto, Japan; 25–28 September 2001).

[27] R. Douence and M. Südholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, Dec. 2002.

[28] S. Drape, O. de Moor, and G. Sittampalam. Transforming the .net intermediate language using path logic programming. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 133–144, New York, NY, USA, 2002. ACM Press.

[29] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of aspectj programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 150–169, New York, NY, USA, 2004. ACM Press.

[30] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[31] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98: Proceedings of the 12th European Conference*

on Object-Oriented Programming, pages 186–211, London, UK, 1998. Springer-Verlag.

[32] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[33] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In P. Tarr, L. Bergmans, M. Griss, and H. Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

[34] R. E. Filman and K. Havelund. Realising aspects by transforming for events. In K. De Volder, K. Mens, T. Mens, and R. Wuyts, editors, *Proc. Workshop on Declarative Meta Programming to Support Software Development*, Sept. 2002.

[35] R. E. Filman and K. Havelund. Source-code instrumentation and quantification of events. In *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, pages 45–49, 2002. Position paper.

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1994.

[37] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 31–44, London, UK, 1991. Springer-Verlag.

[38] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Trans. Inf. Syst.*, 14(3):268–296, 1996.

[39] W. Harrison and H. Ossher. Subject-oriented programming—A critique of pure objects. In *Proc. 1993 Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, Sept. 1993.

[40] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.

[41] K. Havelund and G. Roşu. Sythesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), Grenoble, France*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer Verlag, April 2002.

[42] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In Lieberherr [54], pages 26–35.

[43] D. Hoffman and P. Strooper. State abstraction and modular software develop-

ment. In *SIGSOFT'95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 53–61, 1995.

[44] W. L. Hürsch and C. V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, Feb. 1995.

[45] D. H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 347–349, New York, NY, USA, 1986. ACM Press.

[46] S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. 1979. AT&T Bell Laboratories Technical Report July 31, 1978.

[47] A. Kay. Prototypes vs classes. http://c2.com/cgi/wiki?AlanKayOnMessaging, October 1998.

[48] G. Kiczales, editor. *Proc. 1st Int'l Conf. on Aspect-Oriented Software Development (AOSD-2002)*. ACM Press, Apr. 2002.

[49] G. Kiczales and E. Hilsdale. Aspect-oriented programming. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th Acm Sigsoft Symposium on Foundations of Software Engineering*, page 313. ACM Press, 2001.

[50] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001—Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.

[51] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997.

[52] R. Laemmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In Akşit [3], pages 168–177.

[53] D. Larochelle, K. Scheidt, and K. Sullivan. Join point encapsulation. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.

[54] K. Lieberherr, editor. *Proc. 3rd Int'l Conf. on Aspect-Oriented Software Development (AOSD-2004)*. ACM Press, Mar. 2004.

[55] K. Lieberherr and D. H. Lorenz. Coupling aspect-oriented and adaptive programming. In Filman et al. [32], pages 145–164.

[56] K. J. Lieberherr, I. Silva-Lepe, and C. Xiao. Adaptive object-oriented programming using graph-based customization. *Comm. ACM*, 37(5):94–101, May 1994.

[57] M. F. Lindemans. Norns. http://www.pantheon.org/articles/n/norns.html.

[58] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr. Beyond aop: toward naturalistic programming. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 198–207. ACM Press, 2003.

[59] M. Mezini, editor. *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, New York, NY, USA, Mar. 2005. ACM Press. General Chair-Mira Mezini and Program Chair-Peri Tarr.

[60] G. C. Murphy, R. J. Walker, and E. L. A. Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on Software Engineering*, 25(4):438–455, 1999.

[61] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, M. P. Robillard, A. Lai, and M. A. Kersten. Does aspect-oriented programming work? *Comm. ACM*, 44(10):75–77, Oct. 2001.

[62] A. Okhotin. LR parsing for conjunctive rammars, 2002. Grammars, 5:2 (2002), to appear.

[63] OMG. OMG Unified Modeling Language Specification™. http://www.omg.org/technology/documents/formal/uml.htm, mar 2003.

[64] H. Ossher, W. Harrison, F. Budinsky, and I. Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proc. 7th IBM Conf. Object-Oriented Technology*, July 1994.

[65] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using Hyperspaces. Technical Report 21452, IBM Research Report, Apr. 1999.

[66] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, Dec. 1972.

[67] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[68] J. Postel and J. Reynolds. File Transfer Protocol (FTP). Request for Comments 959, Network Working Group, 1985. ftp://ftp.isi.edu/in-notes/rfc959.txt.

[69] T. A. Proebsting and B. G. Zorn. Tangible program histories. Technical Report MSR-TR-2000-54, Microsoft Research, May 2000.

[70] A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In Akşit [3], pages 11–20.

[71] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 221–230, 2001. (ICSE-23; Toronto, Canada; 12–19 May 2001).

[72] T. W. Reps. Program analysis via graph reachability. In *ILPS*, pages 5–19, 1997.

[73] P. V. Roy. A useful extension to prolog's definite clause grammar notation. *SIGPLAN Not.*, 24(11):132–134, 1989.

[74] E. Scott and A. Johnstone. Reducing non-determinism in reduction modified LR(1) parsers. Technical Report CSD-TR-02-04, Royal Holloway, University of London, Department of Computer Science, Egham, Surrey TW20 0EX, England, January 2002.

[75] E. Scott and A. Johnstone. Generalised regular parsing. In G. Hedin, editor, *Compiler Construction, Proc. 12th Intnl. Conf., CC2003*, volume 2622 of *Lecture notes in computer science*, pages 232–246, Berlin, 2003. Springer-Verlag.

[76] E. Scott and A. Johnstone. Table based parsers with reduced stack activity. Technical Report CSD-TR-02-08, Royal Holloway, University of London, Department of Computer Science, Egham, Surrey TW20 0EX, England, May 2003.

[77] E. Scott, A. Johnstone, and G. Economopoulos. Generalised parsing: some engineering costs. In *To appear in Compiler Construction, Proc. 13th Intnl. Conf., CC2004*, Lecture notes in computer science, Barcelona, Spain, April 2004. Springer-Verlag.

[78] D. Sereni and O. de Moor. Static analysis of aspects. In Akşit [3], pages 30–39.

[79] G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–38, New York, NY, USA, 2004. ACM Press.

[80] B. C. Smith. *On the Origin of Objects*. Bradford Books. MIT Press, 1998.

[81] K. Sullivan, L. Gu, and Y. Cai. Non-modularity in aspect-oriented languages: Integration as a crosscutting concern for AspectJ. In Kiczales [48], pages 19–27.

[82] D. Suvée, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd*

international conference on Aspect-oriented software development, pages 21–29. ACM Press, 2003.

[83] D. Tannen. Discourse analysis. http://www.lsadc.org/fields/index.php?aaa=discourse.htm.

[84] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119, 1999.

[85] J. S. Uhl and R. N. Horspool. Flow grammars - a flow analysis methodology. In P. Fritzson, editor, *CC*, volume 786 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 1994.

[86] M. VanHilst and D. Notkin. Using role components in implement collaboration-based designs. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 359–369, New York, NY, USA, 1996. ACM Press.

[87] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[88] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects, 2003.

[89] R. J. Walker. *Essential Software Structure through Implicit Context*. PhD dissertation, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Mar. 2003.

[90] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In *Proc. 21st Int'l Conf. Software Engineering (ICSE '99)*, pages 120–130, 1999.

[91] R. J. Walker and G. C. Murphy. Implicit context: Easing software evolution and reuse. In D. S. Rosenblum, editor, *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering (FSE-8): Foundations of Software Engineering for Twenty-First Century Applications*, pages 69–78, San Diego, California, USA, 8–10 Nov. 2000. ACM Press.

[92] R. J. Walker and G. C. Murphy. Join points as ordered events: Towards applying implicit context to aspect-orientation. In P. Tarr and H. Ossher, editors, *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*, May 2001.

[93] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT Twelth International Symposium on the Foundations of Software Engineering*, Newport Beach, CA, USA, Octo-

ber/November 2004. ACM Press.

[94] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In R. Cytron and G. T. Leavens, editors, *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 1–8, Mar. 2002.

[95] Wikipedia. Wyrd. http://en.wikipedia.org/wiki/Wyrd.

[96] J. Xu, H. Rajan, and K. J. Sullivan. Understanding aspects via implicit invocation. In *ASE*, pages 332–335. IEEE Computer Society, 2004.

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