Communication History Patterns: Direct Implementation of Protocol Specifications

Robert J. Walker and Kevin Viggers
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{rwalker, viggers}@cpsc.ucalgary.ca

Technical report 2004-736-01

Abstract

The interactions between separated crosscutting concerns and the base modularity of a system must be specified. These specifications include descriptions of the join points in the base code where behaviour is to be added or replaced. At present, the means available for describing join points generally treat each join point in isolation, rather than allowing multiple join points to be related to each other. When the interactions to be specified consist of complex, stateful communication protocols, the protocols must be hand-compiled into a description of isolated join points. As a result, errors can be introduced into the implementation of the protocols, and the intent of each protocol can become obscured.

This paper describes the use of *communication history patterns* to interrelate multiple join points when implementing complex interaction protocols. A practical language for communication history patterns is described, involving the addition of constructs and preprocessing to AspectJ. A case study is discussed that compares the use of this extended language to both a Java implementation and an unextended AspectJ implementation.

1 Introduction

Aspect-oriented programming (AOP) promotes the separation and modularization of the crosscutting concerns in a system [19]. To form a functioning system, the separated concerns and the base modularity of the system must still interact. We can specify this interaction by describing a set of *join points* in the base modularity (either static points or points in the execution) at which to add or replace behaviour. Most join point descriptions refer to properties of individual join points in isolation, without reference to other join points. This is a significant limitation.

The interactions between the separated concerns and the base modularity can involve complex, stateful communication protocols. For example, the type adaptation work of Yellin and Strom [30] requires that state machines be specified for the translation of communication between two modules not originally designed to explicitly interact with each other (e.g., reused modules). Intrusion detection has been performed by monitoring actual program behaviour and comparing it against expected program behaviour, modelled as complex automata [26]. And user authentication in the File Transfer Protocol (FTP) [23] is a crosscutting concern within an FTP server that requires that a particular pattern of commands be received for a session to be authenticated; we examine the details of this protocol in Section 2.

Through descriptions of isolated join points, it is possible to manually translate these complex communication protocols into implementations of interactions. However, such translation is less than satisfactory, as the original intent of the communication protocol can be obscured. This can lead to the introduction of errors when the translation into an implementation is incorrect, and to incorrect evolution when the maintainer fails to understand the connection between the source they are changing and the original communication protocol. In Section 3, we examine this loss of clarity in implementing the FTP user authentication protocol.

In the AspectJ language [18], available join points include method executions, field accesses, and class initializations, among others. A set of join points is specified through the description of a *pointcut*, which is an equivalence class over the set of all join points available in a program. Pointcut designators describe a set of constraints on the properties of individual join points, such as their type (e.g., method executions), their lexical extent (e.g., within the source declared by a given class), or details of the names and objects involved (e.g., that the method execution involve an argument of a particular type). An important exception involves the `cflow` designator and its variants. When conjoined with other designators, `cflow` allows the restriction of a pointcut to those join points that occur in the control flow of any of another set of join points. For example, a pointcut can be declared that captures those field accesses that occur in the control flow of a given method execution—i.e., after that method execution has begun but before it has ended.

In this paper, we examine how the specification of *communication history patterns* can be used as a practical means for the

direct implementation of communication protocols. Communication history patterns generalize the ideas behind the `cflow` pointcut designator. At a conceptual level, the communication history is a record of all the communications occurring in a system (e.g., method calls, field accesses); it is a trace of the system events.[1] Patterns on the communication history can be specified; these patterns make reference to a set of join points and their interrelationships in the structure of the execution, such as constraints on their temporal or causal ordering. Communication history patterns are reminiscent of features provided by a variety of specification techniques, such as trace assertion [6, 16], Gist [11], and the technique of Broy and Stølen [7]. Trace assertion, for example, has been used in specifying a variety of complex communication protocols, including the Internet Protocol [14]. Communication history patterns differ from these techniques in two ways: they provide a means to specify interrelated join points that are to be accessed in external modules, and a means to directly implement these specifications. Communication history patterns provide a bridge from specification to implementation, improving the clarity of intent in the resulting implementation.

In Section 4, we discuss the concepts of communication history patterns in more detail, and describe two implementation approaches: a "strawman" approach that served as a proof of concept, but which would be impractical for most applications; and a "woodenman" approach that is practical, but remains to be optimized. We have applied the strawman implementation in a number of case studies described elsewhere [28, 27]. We describe the application of the woodenman approach to the implementation of the FTP user authentication protocol in Section 5, and compare it to the Java and AspectJ implementations.

Section 6 provides a discussion of issues surrounding communication history patterns. Related work is described in Section 7.

This paper makes three contributions: a description of the concept of communication history patterns, their use as a means of specifying pointcuts involving relatively complex communication protocols, and the implementation of a practical language supporting them.

2 The File Transfer Protocol and User Authentication

The File Transfer Protocol (FTP) defines a set of commands that can be issued by a client for remote file transfer and the appropriate responses by a server. The reaction of a server to most commands differs significantly depending on whether the user of the client has or has not successfully logged into the server—in general, no actions will be taken unless the user has been authenticated. User authentication is thus a crosscutting concern within any FTP server implementation.

[1]The term *trace* is used in two different ways in the literature, both the way we use it here and (roughly) to refer to languages of these traces—sometimes called Mazurkiewicz traces [9].

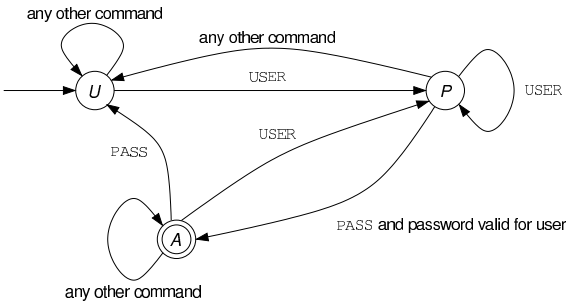

Figure 1: The state machine for authentication implied by RFC 959.

FTP uses two kinds of connections to communicate between client and server, control connections and data connections. A client sends commands to the server via the control connection. The server processes each command and replies with codes indicating the status of the requested operation along with any simple return data. Files are transferred via the data connection. Each connection is stateful and some commands must be sent in an order prescribed by 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 [23]) 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.

Consider the finite state machine for user authentication, shown in Figure 1, that can be derived through a careful analysis of the FTP specification. 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.

Alternative to this finite state machine, we can consider the trace of FTP commands generated as a given FTP session proceeds. The session is authenticated at a particular moment in time 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 (plus a password check) on the session trace that can be expressed as a regular expression (in `grep` syntax):

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

The finite state machine in Figure 1 is exactly that which recognizes the language described by the regular expression. The first time that we attempted to derive the finite state machine representation from the FTP specification, we did it wrong. Despite reviews by multiple colleagues, no one noticed the errors in that original finite state machine; the errors were not detected until an implementation failed to behave as expected and until implementation errors relative to the state machine specification were ruled out. On the other hand, a direct implementation of the regular expression specification via communication history patterns worked as expected without needing to correct the specification (see Section 5).

Below, we describe attempts to implement the user authentication protocol in an FTP server. In Section 3, we see how an implementation in Java suffers most obviously from the scattering and tangling of the crosscutting concern with the base code. However, separating this concern in an AspectJ implementation leaves us with the inability to directly implement the regular expression, forcing us to manually encode the transitions in the finite state machine representation. In Section 5, we see how augmenting AspectJ with communication history patterns allows a direct implementation of the regular expression, resulting in greater clarity of intent within the server implementation.

3 FTP Authentication in Java and AspectJ

We designed and implemented an FTP server providing the Minimum Implementation subset of features from the FTP specification; the implementation was created in Java. This base design (Section 3.1) ignored the presence of the remainder of the FTP specification. The base design was then extended in two versions to add the user authentication feature: one version made use of Java (Section 3.2) and the other made use of AspectJ (Section 3.3). We discuss the results of these implementations in Section 3.4.

3.1 Base Design

The UML class diagram for the base design is shown in Figure 2. A class hierarchy representing different reply codes is not shown. Asterisks indicate singletons. Some dependencies are shown, indicated by dashed arrows. Boldface classes run in their own threads.

An instance of `Server` runs listening for connection attempts by clients on a particular port. When a client connects to the server, an instance of `Session` is created. In turn, the `Session` object creates one instance each of `ControlConnection`, `DataConnection`, and `TransferContext`. The `Session` object is used to provide a convenient handle on the current control connection, data connection, and transfer context.

A `ControlConnection` listens at its port for incoming request messages from the client. Received requests are sent through a chain of objects for parsing and action: `Interpreter`, `CommandFactory`, and a hierarchy of `Command` subclasses. One or more responses are generated along this chain and handed to the `ControlConnection` to be sent to the client. The `DataConnection` is responsible for the formatted transfer of files to and from the client. The `TransferContext` object contains the state specific to the session, such as the address and port of the client.

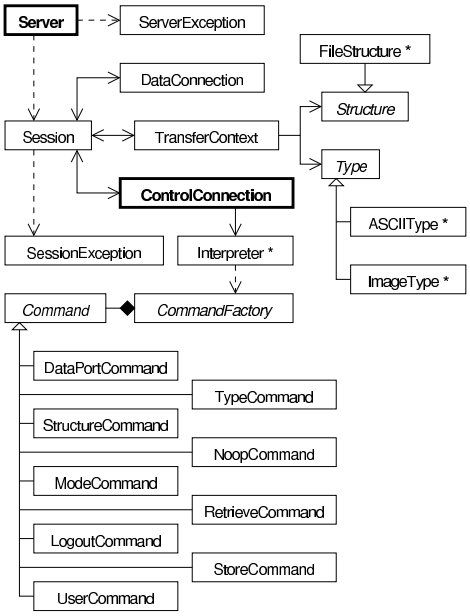

Figure 2: Base design for the FTP server.

3.2 Extension in Java

Authentication required additions to several classes in the base design. A field was needed in the `TransferContext` class to record whether the user had been authenticated, along with getter and setter methods for this state. A `PasswordCommand` was added to the `Command` hierarchy to parse the arguments to the `PASS` command. A class called `Authenticator` was created to perform the actual validity check on a user name/password pair. `PasswordCommand` uses `Authenticator` to perform this validity check, storing the results in the current `TransferContext`. The `UserCommand` had to be altered so as to reply "Password needed" to the client.

Every other command, save `QUIT`, must be authenticated prior to its operation. Therefore, 8 of the 9 subclasses in the `Command` hierarchy had to be 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.

3.3 Extension in AspectJ

The AspectJ extension consisted of a single `Authentication` aspect and two classes. The aspect added a field and getter and setter methods for the user authentication state to `TransferContext`, as in the Java extension. `Authenticator` and `PasswordCommand` classes were added as in the Java extension.

This aspect defined five pointcuts: one for capturing all executions of the `perform` method on any `Command` subclass; one each for capturing such executions specifically on `UserCommand`, `PasswordCommand`, and `LogoutCommand`; and one for capturing executions of commands where the session is not authenticated but needs to be ("invalid" command executions). Examples for `LogoutCommand` and invalid command executions are shown in Figure 3.

```
pointcut logout(String arg, Session s):
  args(arg, s) &&
  execution(* * LogoutCommand.
                perform(String, Session));

pointcut invalid(Session s):
  command(*, s) && !user(*, s) &&
  !password(*, s) && !logout(*, s) &&
  if(!s.getTransferContext().
      isAuthenticated());
```

Figure 3: Example pointcuts declared in the AspectJ-extended version.

Three pieces of `around` advice were declared in the aspect (see Figure 4). The first of these alters the behaviour of `UserCommand`: the receipt of `USER` causes the session to become unauthenticated (state P) and a "Password needed" reply is sent. The second advice alters invalid command executions so that the session becomes unauthenticated (state U) and 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. If there is a user name recorded in the session's `TransferContext`, the session is assumed to be awaiting a password (state P); if the user name and password correspond, the session becomes authenticated (state A). Otherwise, the session becomes unauthenticated (state U) and the user name is unset.

```
// Switch to awaiting password and replace the
// response to the USER command to ask for password
void around(String arg, Session s, Reply r):
  cflow(user(arg, s)) && args(r) &&
  call(* * ControlConnection.send(Reply)) {
    TransferContext tc = s.getTransferContext();
    tc.setAuthenticated(false);
    proceed(arg, s, new NeedPasswordReply());
}

// Switch to unauthenticated and replace the
// response to say session is unauthenticated
void around(Session s): invalid(s) {
  ControlConnection cc =
    s.getControlConnection();
  TransferContext tc = s.getTransferContext();
  tc.unsetUserName();
  cc.send(new NotLoggedInReply());
}

// Check password against immediately preceding
// USER; authenticated if match; unauthenticated
// otherwise
void around(String pwd, Session s, Reply r):
  cflow(password(pwd, s)) && args(r) &&
  call(* * ControlConnection.send(Reply)) {
    TransferContext tc = s.getTransferContext();
    if(!tc.hasUserName() ||
       !Authenticator.
       isValid(tc.getUserName(), pwd)) {
      tc.setAuthenticated(false);
      proceed(pwd, s, new LogInFailedReply());
    } else {
      tc.setAuthenticated(true);
      proceed(pwd, s, r);
    }
}
```

Figure 4: Advice declared within the AspectJ-extended version.

3.4 Results

The Java-extended version scatters and tangles the responsibilities for handling authentication throughout the system. The system assumes that all its parts will correctly make use of the getter and setter methods on `TransferContext` to determine or alter the recorded authentication state of the session. Each event potentially causing a transition in the finite state machine representation is parsed by a different part of the system; each part must then locally decide if a transition is warranted and must cause this transition to occur by invoking the appropriate methods on `TransferContext`. 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.

In the AspectJ-extended version, the code that records a change in state of the session remains scattered and tangled. It is scattered amongst three pieces of advice, and it is tangled with the code that implements the corresponding changes to replies returned to the client. As a result, the correspondence between the implementation and the specification of the FTP authentication protocol is obscure.

The AspectJ-extended version reduces the scattering and tangling of the crosscutting user authentication concern present in the Java-extended version. Nevertheless, the clarity of intent remains poor in this implementation.

4 Communication History and Patterns

The state of a system is the product of the computations taking place within it. These computations result from the communications that occur between the parts of the system. The decisions that are made within the system generally depend on whether the state falls within one equivalence class or another. For example, when a branch is taken within a procedure because the value of a particular variable is positive, the remainder of the state of the system is ignored; all system states where that variable is positive are equivalent.

We can specify behaviour relative to an equivalence class of states. In expressing such an equivalence class, our goal should be to describe exactly the relevant properties of that equivalence class. One way in which an equivalence class of states can be expressed is as a pattern within the communication history of the system.

The communication history is a trace of communication events that is both collected and queried as the system executes. At any given moment during the system execution, this trace includes all the events that have occurred in the system from its start to the present execution point. Context of the communication events can be exposed by retrieving information about arguments and result values from the history.

We consider an event to be an observable, instantaneous occurrence of behaviour. Events within a trace can be ordered temporally, causally, or both. An event class is a set of events that are (or would be) identical save for when they occur. The universe of event classes available in a given system is both a flexible and important choice. Some possibilities include method entries and exits, field accesses, object creations, etc.

Below, we describe two implementations of communication history patterns. In Section 4.1, the model suggested here is implemented verbatim; however, this results in serious inefficiencies that make the approach impractical in most situations. In Section 4.2, certain optimizations are utilized to avoid these inefficiencies, resulting in a practical technique for using communication history patterns.

4.1 A Strawman Implementation

As a simple, initial realization of the idea of communication history patterns, we implemented the model described above quite literally. This implementation was created as part of the prototype version for the IConJ language, a language intended to ease software evolution and reuse [27]. IConJ source code is fed to a preprocessor for translation to Java source code. Further details of IConJ lie outside this paper.

A simple API for accessing communication history was available in the form of a set of operations on a `History` class. Many of these operations return instances of an introspective `Call` class, detailing the method having been called and the arguments passed within the call. Some example operations are listed in Figure 1. It is important to understand that the arguments within a message are saved within the communication history as-is. In Java, objects are passed by reference; therefore, it is those references that are stored, not the object state behind those references.

Part of API to `History`:

`lastInstancePassed(`c`)`	returns instance of class c that was most recently passed as argument
`lastCall(`P`)`	returns last call conforming to pattern P (i.e., instances of `Thread`, `Class`, etc.)
`lastCallInCFlow(`P`,`k`)`	returns last call conforming to pattern P occurring as descendent of `Call` k

Part of API to `Call`:

`getPredecessor()`	returns instance of `Call` that happened most recently before this `Call`
`getArgument(`i`)`	returns ith argument passed in this `Call`
`precedes(`k`)`	returns `boolean` indicating whether this `Call` occurred prior to `Call` k

Table 1: Sample of API to communication history in the strawman approach.

The IConJ preprocessor instruments the source code to support the communication history API; the prototype performs this instrumentation in a naive (and incomplete) fashion. Every method entry and method exit causes information to be stored into a data structure that is a set of trees (one per thread) conforming to the nested structure of entries and exits that occur at run-time. This data structure also records the temporal sequence of entries and exits, irrespective of the thread in which each occurs. Field accesses and method calls (as separate from entries) are ignored. Individual operations in the API allow the developer to either worry about or ignore threading issues as appropriate to a given situation.

There are four troubles with this approach. (1) Every method in the source code must be instrumented to store information about its entry and exit to the `History` data structure. This results in code bloat. (2) Each of these instrumented points must be executed resulting in slowdown of the system. (3) Data on each entry and exit must be stored forever. The longer the execution, the more storage space that is required to maintain the communication history; garbage collection is defeated. (4) Some queries require time to check that is proportional to the size of the communication history. The longer the execution of the system, the slower some queries would take to run.

While this strawman sufficed as a proof of concept, it is clearly insufficient as a practical approach. A better approach can be achieved by realizing that only certain events within the system need be recorded and then that the data collected there often does not need to be kept forever.

4.2 A Woodenman Implementation

Rather than instrument every potential event-generating point in the code, we need only instrument those that potentially generate events of interest to the patterns on communication history actually being specified.

For example, in the regular expression describing the authenticated state for an FTP session, we see mention of `USER` and `PASS`—events representing receipts of these commands must therefore be noted. That regular expression also makes use of the term `[^USER,PASS]` which indicates any event other than receipt of a `USER` or `PASS` command. However, we must define a universe under which this complement can be calculated. The implication in this case is that we are interested in any events involving FTP commands, and nothing else.

Making this optimization allows for the reduction of the instrumentation points in the code and of the rate of growth of the communication history data being collected. However, this data can still grow without bound. To overcome this obstacle, we must recognize that each communication history pattern can be realized as an automaton. Events cause transitions in this automaton and then can be discarded, ordinarily. The exception is when context must be exposed: the system may need to retrieve arguments or result values associated with events that match the pattern. Therefore, any candidate to replace the strawman approach should guarantee an upper bound on the quantity of data that a given automaton must store. Ideally, this bound should be proportional to the size of the pattern being sought.

It is necessary to select a class of languages in which the patterns of interest will be expressed. Too weak, and it would not be possible to express the patterns for which we typically search. Too powerful, and the patterns are liable to be too hard to use for simple patterns and too hard for tools to analyze. For the woodenman approach, we have settled on context-free languages (CFLs) for expressing the patterns. CFLs combine a reasonable level of expressibility, usability, and analyzability. Most importantly, CFLs permit us to express proper nesting, which is key for expressing patterns over the trees representing communication history. CFLs have also allowed us to apply the large body of knowledge surrounding parser and compiler technology.

The language we have developed to make use of the woodenman approach extends AspectJ with constructs to specify context-free grammars over primitive events and traces. These new constructs provide a means of collecting and forwarding context along the matching execution paths, as well as conditionally limiting the application of advice to occurrences of communication history patterns. The syntax and informal semantics of this language are described in Section 4.2.1.

We have developed a prototype preprocessor that translates this extended AspectJ source code into non-extended AspectJ. The output of this preprocessor includes generated aspects which faithfully transform communication history patterns, as specified with our additional constructs, into simple AspectJ advice declarations. This advice acts to provide input to one or more generated parsing automata. These automata are used to dynamically monitor the execution trace recognizing and optionally construct semantic structures derived from the pattern grammar. The details of the translation process are described in Section 4.2.2.

4.2.1 Syntax and Informal Semantics

A tracecut describes a set of traces in the execution of the system, analogous with the connection between pointcuts and join points. Three significant constructs are introduced by our AspectJ extension to describe and use tracecuts: primitive tracecut designators, ordered tracecut designators, and history queries. Each of these constructs is described below. Figure 5 shows the syntax for these extensions. Rules beginning with a capital letter correspond to the standard AspectJ or Java interpretations.

history_pcd	→	**history** (*tracecut*)
tracecut	→	*primitive_tracecut* | *named_tracecut_ref*
		| *ordered_tracecut* | **^** | **$** | (*tracecut*)
		| **[** *tracecut* **]** | *tracecut* **+** | *tracecut* *****
primitive_tracecut	→	**entry** (*pointcut*) | (**exit** (*pointcut*)
		[(**throwing** | **returning**) *[* (*Formal*) *]])*
named_tracecut_ref	→	*Id* (*Actuals*)
ordered_tracecut	→	(*tracecut*)* *[* {**:** *semantic_action* **:**} *]*
named_tracecut	→	*abstract_tracecut* | *concrete_tracecut*
abstract_tracecut	→	**abstract** *[Modifiers]*
		tracecut *Id* (*Formals*) **;**
concrete_tracecut	→	*[Modifiers]* **tracecut** *Id*
		(*Formals*) *[* { *semantic_decls* } *]*
		::= *disjunct* **;**
disjunct	→	*ordered_tracecut* (**|** *ordered_tracecut*)*

Figure 5: Extensions to the AspectJ grammar.

Communication history patterns in advice A tracecut can be used in advice by supplying it as the argument to a `history` pointcut designator, which we added to the extended AspectJ language. The associated advice is performed only if the specified pattern has occurred in the communication history. The tracecut is matched greedily against the communication history, meaning that the earliest occurrence of the specified pattern will match.

Primitive tracecuts Primitive tracecut designators define the lexemes of communication history patterns, capturing individual events in the execution trace.

Two primitive tracecut designators are defined in our extended AspectJ language. Entrance into a join point can be captured through the use of the `entry` tracecut designator, while exits from a join point can be captured through the use of the `exit` tracecut designator. Both of these operate on a standard AspectJ pointcut, which can expose context through formal parameters in the standard AspectJ fashion.

Consider the example shown in Figure 6. A tracecut named `foo` is declared in the form of a production rule that reduces to a primitive tracecut. This primitive tracecut specifies events involving

entry to the method `A.foo`; the argument passed to this method is exposed in the formal parameter `i`. The `foo` tracecut is then used within `before` advice. The `somePC()` pointcut (the declaration of which is not shown) specifies the join points where the `before` advice should apply, as long as the method `A.foo` has previously been entered. This advice is then able to make use of the value bound to the formal parameter. Rather than name this tracecut, we could replace the reference to `foo` within the `history` pointcut designator with the explicit right-hand-side of the production rule.

```
tracecut foo(int i) ::=
  entry( execution(void A.foo(int)) && args(i) );

before(int i): somePC() && history( foo(i) ) {
  // Advice code making use of i
}
```

Figure 6: An example of a primitive tracecut.

Ordered tracecuts Primitive tracecut designators are not very expressive in isolation. When combined with other tracecuts however, ordering relationships between events can be stated. We call these statements ordered tracecuts. By exposing context, an ordered tracecut allows one to construct an *information bridge* between remote points in the execution trace without explicitly stating these pathways within the source code.

Ordered tracecuts are declared similarly to primitive tracecuts. A named tracecut can declare a reduction to a sequence of primitive tracecuts or other named tracecuts; this reduction can be recursive. Ordered tracecuts can optionally expose context associated with their occurrences—state such as arguments, result values, and other relevant information available at individual join points along the path of the trace. Such context can be exposed to advice on the tracecut occurrence through the use of formal parameters.

In our woodenman prototype, we limit the relationships between events to those of immediate dominance (e.g., A causes B) and precedence (e.g., A precedes B), the relationships expressible with context-free languages.

Consider a simple example of an ordered tracecut, shown in Figure 7. This tracecut specifies a communication history pattern where we want to recognize a sequence of calls to `foo`. Specifically, we would like to locate a third (non-recursive) call in a row to `foo`. Although an admittedly contrived example, it does illustrate the expressiveness of communication history patterns.

```
tracecut threeFoos(Object caller) ::=
  foo() foo() foo(caller);
```

Figure 7: A simple sequencing example.

Consider a second example of an ordered tracecut, as shown in Figure 8, that simulates the effect of the `cflow` pointcut designator of AspectJ. This example begins by specifying tracecuts that use `entry` and `exit` to capture the beginning and ending events of a pointcut named `somePC` (`somePC` might be a method execution, for example). We define an ordered tracecut, named `complete`, that represents a completed entrance–exit pair of `somePC`, including any nested completions. Finally, we define a second ordered tracecut, named `inCflow`, that requires one unmatched entry to `somePC`. The dollar sign indicates the current end of the communication history, and forces events to continue to be matched against the pattern throughout the execution of the system. The `before` advice is then performed before any execution of `A.foo` that is in the control flow of `somePC`.

```
tracecut enterCflow() ::=
  entry( somePC() );

tracecut exitCflow()  ::=
  exit( somePC() );

tracecut complete() ::=
  enterCflow() complete()* exitCflow();

tracecut inCflow() ::=
  enterCflow() complete()* $;

before(): history( inCflow() ) &&
          execution(void A.foo()) {
  // Advice code
}
```

Figure 8: Ordered tracecuts simulating `cflow`.

Semantic declarations and action blocks The context exposed within tracecuts can be assigned or manipulated in an optional semantic action block of Java code. For primitive tracecuts this typically involves modifying the state drawn from the join point before assigning it to a formal parameter. Such computations can make use of temporary local variables, which are declared in a block on the left-hand-side of the production rule for a named tracecut declaration. These local variables can be bound to context exposed on the right-hand-side of the production rule.

For example, Figure 9 shows a situation in which a tracecut on the right-hand-side of a production exposes context of the wrong type for our purposes. The `entry` primitive tracecut exposes the `Integer` argument passed to calls of `A.foo`; however, we need the named tracecut that we are declaring (namely `foo`) to expose a value of type `int`. Thus, we declare a local variable `integer` of type `Integer` to capture the argument exposed from the call to `A.foo`, and then convert this argument to an `int` value within the semantic action block.

It is also possible to apply semantic actions to conditionally reject event occurrences. In the example, we enforce a semantic constraint on the class of events selected by the `foo` tracecut: `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.

```
tracecut foo(int i) { Integer integer; } ::=
  entry( call(void A.foo(Integer))
         && args(integer) )
  {:
     i = integer.intValue();
     if(i & 1) // bit operation
       fail; // reject this occurrence
  :};
```

Figure 9: An example using a semantic action to translate exposed context.

4.2.2 Translation to AspectJ Source

Our preprocessor operates on AspectJ aspects extended with our additional syntax. The preprocessor translates constructs in the extended syntax to standard AspectJ constructs.

Parser generation For each `history` pointcut designator for which advice is applied, the preprocessor attempts to construct a pushdown automaton. This automaton will parse the context-free grammar specified by the target tracecut given as the argument to that `history` pointcut designator. These automata are implemented as standard table-driven LR parsers [20, 2, 4].

The target tracecut can make reference to other tracecuts. The transitive closure of these references defines the production rules for the automaton implementing the target tracecut. The leaves of the transitive closure consist of primitive tracecuts.

Each primitive tracecut is translated into AspectJ advice; each `entry` becomes `before` advice, while each `exit` becomes `after` advice. During the execution of the woven system, the occurrence of an event matching a primitive tracecut will cause one or more tokens to be generated, each of which will be sent to a different automaton for recognition.

The preprocessor applies token-generation advice for each parser (and hence, each communication history pattern) even if the pointcuts for this advice are not disjoint across all parsers. This may involve some code bloat, but token generation for each parser does not need to be ordered. We have conducted initial investigation into the use of general LR recognition coupled with the Schrödinger's Token [3] as a means of allowing a join point to simultaneously be a member of multiple terminal classes. Such an idea remains to be incorporated in future approaches.

Each `history` pointcut designator is translated into an `if` pointcut designator that tests the state of the corresponding automaton. If a given communication history pattern has occurred, the corresponding automaton will be in an accepting state; otherwise, it will not be in accepting state.

Semantic stack The semantic stack of each parser provides a means of storing the exposed context of a pattern as it is recognized. Primitive tracecuts obtain state directly from their join point bindings; the tokens generated at each join point store the exposed state of the event and carry it to the parser. More complex traces derive their state from the individual tracecut occurrences in a trace pattern. When an ordered tracecut is recognized by the parser, operations on the semantic stack are executed to combine state placed on the stack.

Like primitive tracecuts, ordered tracecuts may have associated semantic actions. For primitive tracecuts, these actions determine how state exposed at the join point is assigned to the tracecut state and even if the primitive tracecut should be recognized. These primitive semantic actions execute every time an occurrence of the primitive event class is encountered in the execution trace. With ordered tracecuts, semantic actions are executed immediately after the tracecut pattern is recognized. Ordered tracecuts use this opportunity to manipulate state from its constituent parts. Tracecuts may also reject the occurrence of the pattern based on some semantic condition, which causes the parser to return to its initial state.

Consider the example shown in Figure 10. We wish to advise the eventual occurrence of the creation of a `String` object with the value `"hello"` followed by the eventual occurrence of the creation of a `String` object with the value `"world"`. The string that is exposed by the `helloWorld` tracecut (`hw`) is combined from state associated with occurrences of the `hello` and `world` tracecuts. At some point in the execution, a `String` is created with the value `"hello"`; a token storing this state is passed to some automaton and stored on its semantic stack. Later in the execution, an instance of `String` is created with the value `"world"`; this similarly causes a token to be sent to the automaton and stored on its semantic stack. When the correct sequence of these events is recognized, the semantic action of `helloWorld` is executed. The local variables `h` and `w` are bound to state removed from the stack and concatenated to form a new string, which is itself then pushed on to the semantic stack.

```
tracecut hello(String s)::=
  exit( call(String.new(String)) && args(s) &&
        if(s.equals("hello") );

tracecut world(String s)::=
  exit( call(new(String)) && args(s) &&
        if(s.equals("world") );

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

Figure 10: Matching `hello world` via the semantic stack.

4.2.3 Issues

There remain a number of questions and issues that will need to be addressed in moving from the woodenman approach to better solutions. We examine these here.

Events versus join points Rather than extend AspectJ with communication history patterns, we could have chosen to extend another, existing language or create a new one altogether—indeed

```
pointcut exec(): execution(void A.foo());

before(): history( entry(exec()) ) &&
          exec() {
  // Do something before A.foo() executes
}

after(): history( exit(exec()) ) &&
         exec() {
  // Do something after A.foo() executes
}
```

Figure 11: An example where differences between events and join points can be problematic.

the strawman implementation was created in the prototype IConJ language [27]. We chose to extend AspectJ both to take advantage of its existing features and to simplify comparison between source code using the extended and non-extended languages.

However, this combination is imperfect. Communication history operates 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 (such as `before`, `after`, or `around`) and pointcut that determines when an "event" really happens.

This difference can lead to subtle problems. Consider the example shown in Figure 11. As the system executes, there will come a point where the system event trace matches the pattern: `^ exec()`. It is unclear whether the advice shown will execute here. Strictly speaking, the entry event should not have occurred when the first `exec()` join point is encountered and so this advice should not execute then. But in the symmetric situation, the `after` advice definitely would. These subtleties, while interpretable by a tool, are likely to cause confusion amongst developers. It would be preferable to define a simpler, consistent event/join point model.

Universe of tokens In our woodenman implementation, the universe of tokens that are sent to an automaton depends on the tracecuts used to define that automaton. Only the primitive tracecuts in the transitive closure of the target tracecut will send tokens to the corresponding automaton. This has pros and cons. On the one hand, this allows us to simplify the expression of communication history patterns: we do not need to make mention of token types that the pattern does not care about. On the other hand, we then need to make explicit mention of any tokens that we definitely do not want to occur.

For example, consider the actual trace: `a b c`. If we specified a target tracecut consisting of the sequence `a c`, this pattern would match this trace: our automaton would not care about `b` events.

We should provide a means to specify the universe of tokens to be considered by an automaton when it differs from those explicitly mentioned in the transitive closure of the target tracecut. Complementation would then be unambiguously defined and a complement operator could be added to the syntax. These are straightforward extensions, but remain to be done.

Specification of interest in particular event classes has another advantage: backwards compatibility in the face of extensions to a language's join point model. Some pointcut designators (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 pointcut designators 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. Our woodenman approach is a hybrid of event-based and join-point-based, so it does not have this advantage.

Generic patterns In Figure 8, we simulated the `cflow` pointcut designator via communication history patterns. Such a simulation would need to be cut-and-pasted for every different `cflow` in which we were interested. Instead, adding the ability to define parameterized tracecuts would permit the encoding of commonly-occurring patterns. These generic patterns would effectively define new operators in terms of structural and temporal properties of event traces.

Multi-threading While the strawman approach explicitly allowed for communication history patterns to be matched within a single thread or across multiple threads, the woodenman implementation creates automata strictly on a per-thread basis.

In practice, we have very rarely used the multi-threaded queries in the strawman approach. However, the woodenman implementation can be extended. The syntax could provide an explicit `multithread_history` pointcut designator; the thread in which each event occurred could be exposed through formal parameters.

The use of communication history does not allow one to magically avoid synchronization. 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 might be possible. Such issues require further research.

Further optimization Static analysis could be conducted to further optimize the approach. Some sub-patterns within a communication history pattern could be compared against the control-flow graph of a system to recognize infeasible paths or definite sequences. The number of join points at which token-generating instrumentation is necessary could be reduced, as could the complexity of the associated parser automata. Such optimizations could operate on principles that extend the work of Sereni and de Moor [25].

5 FTP Authentication via Communication History Patterns

We now examine how our extended AspectJ language can be applied to the implementation of the user authentication protocol for the FTP server. Communication history patterns promise to make the implementation of protocol specifications more direct, thereby improving the clarity of intent behind such an implementation.

Our augmented `Authentication` aspect (Figure 12) contains three abstract tracecuts related to the authentication protocol. These tracecuts represent executions of `UserCommand`, `PasswordCommand`, and all other commands requiring authentication (`notUserNotPass`), respectively.

Using these three tracecuts we specified an ordered tracecut `isAuthenticated` that corresponds closely to the regular expression for the authenticated state: `USER PASS [^USER,PASS]*`. This tracecut uses semantic actions to reject patterns where an invalid user name/password pair is detected. Figure 13 shows the `isAuthenticated` tracecut, hiding the parts of the declaration present for collecting and manipulating context. The similarity between the `isAuthenticated` tracecut and the regular expression for the authenticated state is evident.

```
abstract tracecut user(String name);
abstract tracecut pass(String pwd);
abstract tracecut notUserNotPass();

tracecut isAuthenticated()
  { String name, pwd; } ::=
  ( user(name) pass(pwd)
    {:
      if(!Authenticator.isValid(name, pwd))
        fail;
    :} ) notUserNotPass()* $;

void around(Session s):
  commandPC(s) && !userPC() &&
  !passPC() && !logoutPC() &&
  !history(isAuthenticated()) {
    ControlConnection cc =
      s.getControlConnection();
    control.send(new NotLoggedInReply());
}

void around(Reply r):
  args(r) &
  call(* * ControlConnection.send(Reply)) &&
  cflow(passPC()) &&
  !history(isAuthenticated()) {
    proceed(new LogInFailedReply());
}
```

Figure 12: The authentication protocol in AspectJ extended with communication history patterns.

```
tracecut isAuthenticated() ::=
  user() pass() notUserNotPass()* $;
```

Figure 13: The tracecut for the authenticated state, hiding context exposure declarations.

As in the original AspectJ authentication solution, we still had to advise FTP commands to ensure a correct response. The first piece of advice replies with "Not logged in" whenever a command other than those exempt from authentication (`USER`, `PASS`, or `QUIT`) is issued in a session that has not been authenticated. The second piece of advice informs the user when a invalid user name/password combination has been entered. Calls to `ControlConnection.send`, issued from within the `PasswordCommand.perform`, are altered to reply "Login failed" whenever a failed attempt occurs.

We have used abstract tracecuts in our example to highlight that the pattern specification is reusable. For brevity, we have omitted the definitions of `commandPC`, `userPC`, `passPC`, and `logoutPC` pointcuts.

5.1 Comparison

We were able to represent the regular expression for user authentication directly with the use of our extended AspectJ language. Authentication becomes a simple query on the occurrence of the tracecut, resulting in localization of the state specification as compared to the solution in standard AspectJ (Section 3).

In the standard AspectJ solution, `TransferContext` played a significant role in holding the state of the protocol—it was a mediator in the interaction between multiple aspects without controlling how that mediation should occur. The extended version alleviates this coupling; the required state for determining authentication is implicitly stored and manipulated through the mechanisms of communication history.

Authentication still involves the identification of the points in the execution where our primitive tracecut events should be emitted. In the standard AspectJ solution, we had to describe the behaviour at individual join points, exposing context and storing state these points could be interrelated indirectly. Using communication history patterns we were able to state these relationships directly and concisely.

6 Discussion

In this section, we consider a number of questions and outstanding issues surrounding the use of communication history patterns.

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 [30]. And demanding a formal specification of every protocol to be used is not too realistic. FTP does specify the key properties of importance, and that was considered sufficient by its authors.

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. Communication history patterns permit such flexibility by specifying only the paths of importance.

The woodenman approach to communication history patterns would not work in some cases of dynamic loading. Consider a dynamically loaded module that included some communication history patterns. Unless the system was aware of these pattern specifications ahead of time, no automaton could have been created to parse events in the system prior to the module being loaded. Research is needed into the effects of such an "event horizon"; it is possible that complete access to the communication history is not always needed. Note however that patterns specified by the rest of the system would continue to operate as desired if the correct event instrumentation were added into the dynamically-loaded module at load time, since that module could not have caused any events prior to its loading.

We have chosen to specify communication history patterns via context-free grammars. One might ask if this is sufficient. However, the pattern does not need to operate in isolation; it is used to retrieve information that can be fed into a Turing-complete computation—namely that defined by the advice applied to it. Thus, the question becomes one of whether greater expressibility in the pattern specification is worth the attendant added complexity of expression and difficulty in analysis. More expressive formalisms tend to impose penalties on the runtime monitoring of the execution and complicate the collection and synthesis of semantic information. Our investigations suggest that a slightly better choice than context-free grammars might be conjunctive grammars [22]—context-free grammars augmented with conjunction. Expressibility would be improved through conjunctive grammars without significantly increasing complexity or decreasing analyzability; however, this extension requires further research.

7 Related Work

The idea of communication history was first introduced in connection with an approach, called implicit context, to ease software evolution and reuse through an advanced separation of concerns mechanism [28]. This idea was advanced as being an extension to aspect-oriented programming in a position paper describing a hypothetical `morerecent` pointcut designator [29].

A variety of specification techniques make use of event traces or historical references, such as trace assertion [6, 14, 16], Gist [11], and the stream approach [7]. Such ideas are also used in validation [12], verification [15], and performance optimization [5].

Traces were a mechanism proposed by Kiczales that would allow a developer to explicitly attach to an object a description of the execution path taken by that object [17]. This mechanism allowed polymorphic instantiation. It is not a means for specifying and reacting to more general equivalence classes of system state.

Reiss and Renieris have described a technique for compressing voluminous execution traces as they are collected, through context-free grammar encoding and construction of automata [24]. These ideas are reminiscent of the woodenman approach to collecting and using communication history.

Lacey and de Moor [21] proposed describing conditions on rewrite rules through the use of temporal logic. We initially attempted to make use of several temporal logics to express communication history patterns. While they could express general properties, they were not well amenable to stating queries on the history. Douence *et al.* have advocated an event-based approach to aspect-oriented programming [10]. This work was later extended to make partial use of sequences of events, applied to expressing interactions between base code and separated concerns in an operating system kernel [1]. This work builds atop the rewrite rules proposed by Lacey and de Moor.

Coady *et al.* have described how describing an ordering of join points can be useful in separating crosscutting concerns in an operating system [8].

State abstraction has been promoted as a mechanism for specifying behaviour in modular software development [13]. The ideas are similar to those behind the use of communication history patterns to express equivalence classes of system state; however, state abstraction has not been applied to specify interactions between base code and separated, crosscutting concerns.

Type adaptation via the specification of translation protocols through state machines has been investigated [30], but the authors of this work indicate that it is difficult to create these state machines. The approach has not been extended beyond the interactions between two modules—not sufficient to deal with crosscutting concerns.

8 Conclusions

We must specify the interactions between aspects and base modularity to form functioning systems. Specifications utilizing isolated join points can be created, but result in implementations that scatter and tangle complex communication protocols across multiple pieces of advice and possibly multiple aspects.

We have presented the model of communication history patterns to permit specification of join point interrelationships. Communication history is conceptually a trace of all the communication events that have occurred during the execution of a system until the present moment; communication history generalizes the `cflow`

pointcut designator. We have implemented an approach for reducing the amount of instrumentation needed to collect communication history and store it, to a practical level. This approach makes use of context-free grammars to specify the patterns of interest and parser technology to recognize these patterns during system execution. Further optimizations are possible.

There have been many informal calls for a mechanism to order join points. Communication history patterns are one way to achieve these desires. Others are liable to be promoted. We have found communication history patterns to be useful to detect behavioural patterns of interest, both in the case studied here and those we have studied elsewhere [28, 29, 27]. However, one might be interested in enforcing more general properties, in other situations. If history teaches us anything, it is that no single approach is likely to be ideal for every purpose.

References

[1] R. Åberg *et al.* Evolving an OS kernel using temporal logic and AOP. Position paper in *Int'l Wkshp. on Aspects, Components, and Patterns for Infrastructure Software*, held at the Int'l Conf. on Aspect-Oriented Softw. Dev., 2003.

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

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

[4] J. Aycock *et al.* Even faster generalized LR parsing. *Acta Informatica*, 37(9):633–651, 2001.

[5] T. Ball and J. Larus. Optimally profiling and tracing programs. In *Proc. ACM Symp. Principles Prog. Lang.*, pages 59–70, 1992.

[6] W. Bartussek and D. Parnas. Using assertions about traces to write abstract specifications for software modules. In *Information Systems Methodology*, pages 211–236, 1978. LNCS 65.

[7] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer-Verlag, 2001.

[8] Y. Coady *et al.* Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proc. Joint Euro. Softw. Eng. Conf. and ACM Symp. Foundations Softw. Eng.*, pages 88–98, 2001.

[9] V. Diekert and G. Rozenberg, eds. *The Book of Traces*, World Scientific, 1995.

[10] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Metalevel Architectures and Separation of Crosscutting Concerns*, pages 170–186, 2001. LNCS 2192.

[11] M. Feather. Reuse in the context of a transformation-based methodology. In T. Biggerstaff and A. Perlis, editors, *Software Reusability*, volume 1, pages 337–359. Addison–Wesley, 1989.

[12] M. Felder and A. Morzenti. Validating real-time systems by history-checking TRIO specifications. In *Proc. Int'l Conf. Softw. Eng.*, pages 199–211, 1992.

[13] D. Hoffman and P. Strooper. State abstraction and modular software development. In *Proc. ACM Symp. Foundations Softw. Eng.*, pages 53–61, 1995.

[14] D. Hoffman. The trace specification of communication protocols. *IEEE Trans. Comput.*, 34(12):1102–1113, 1985.

[15] W. Howden and G. Shi. Linear and structural event sequence analysis. In *Proc. Int'l Symp. Softw. Testing and Analysis*, pages 98–106, 1996.

[16] R. Janicki and E. Sekerinski. Foundations of the trace assertion method of module interface specification. *IEEE Trans. Softw. Eng.*, 27(7):577–598, 2001.

[17] G. Kiczales. Traces (a cut at the "make isn't generic" problem). In *Object Technologies for Advanced Software*, pages 27–43, 1993. LNCS 742.

[18] G. Kiczales *et al.* An overview of AspectJ. In *Proc. Euro. Conf. Obj.-Oriented Prog.*, pages 327–353, 2001. LNCS 2072.

[19] G. Kiczales *et al.* Aspect-oriented programming. In *Proc. Euro. Conf. Obj.-Oriented Prog.*, pages 220–242, 1997. LNCS 1241.

[20] D. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[21] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In *Proc. Int'l Conf. Compiler Construction*, pages 52–68, 2001. LNCS 2027.

[22] A. Okhotin. LR parsing for conjunctive grammars. *Grammars*, 5(2), 2002. To appear.

[23] J. Postel and J. Reynolds. File Transfer Protocol (FTP). Request for Comments 959, Network Working Group, 1985.

[24] S. Reiss and M. Renieris. Encoding program executions. In *Proc. Int'l Conf. Softw. Eng.*, pages 221–230, 2001.

[25] D. Sereni and O. de Moor. Static analysis of aspects. In *Proc. Int'l Conf. Aspect-Oriented Softw. Dev.*, pages 30–39, 2003.

[26] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symp. Security and Privacy*, 2001.

[27] R. Walker. *Essential Software Structure through Implicit Context*. PhD thesis, University of British Columbia, 2003.

[28] R. Walker and G. Murphy. Implicit context: Easing software evolution and reuse. In *Proc. ACM Int'l Symp. Foundations Softw. Eng.*, pages 69–78, 2000.

[29] R. Walker and G. Murphy. Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. Position paper in *Adv. Separation of Concerns Wkshp.*, held at Int'l Conf. Softw. Eng., 2001.

[30] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Trans. Prog. Lang. and Sys.*, 19(2):292–333, 1997.