Implementing Protocols via Declarative Event Patterns

Robert J. Walker
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
rwalker@cpsc.ucalgary.ca

Kevin Viggers
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
viggers@cpsc.ucalgary.ca

ABSTRACT

This paper introduces *declarative event patterns* (DEPs) as a means to implement protocols while improving their traceability, comprehensibility, and maintainability. DEPs are descriptions of sequences of events in the execution of a system that include the ability to recognize properly nested event structures. DEPs 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. DEPs are automatically translated into the appropriate instrumentation and automaton for recognizing a given pattern. Support for DEPs has been implemented in a proof-of-concept extension to the AspectJ language that is based on advanced compiler technology. A case study is described that compares the use of DEPs in the implementation of a protocol (FTP user authentication) to the use of a set of other approaches, both object-oriented and aspect-oriented.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software development, software selection*; I.5.5 [**Pattern Recognition**]: Implementation.

General Terms: Design, Human Factors, Languages.

Keywords: Aspect-oriented programming, event patterns, context-free grammars, traceability, comprehensibility, maintainability, instrumentation, parsing, context-sensitive join points.

1. INTRODUCTION

Every software system needs to provide and to adhere to protocols. Protocols may take the form of static interfaces between objects, modules, machines, etc., or of specifications of the sequences of actions or communications that can occur legally or illegally. Examples of protocols occur in every system, ranging in complexity from the public interface to a class, to the collaboration between a set of roles in a design pattern, to the permitted sequences of commands and replies in a communication protocol, such as FTP [22].

In realizing protocols, their specifications must be implemented in combination with other functionality in a system. Details of the specifications become scattered and tangled with the details of the other requirements for the system, resulting in the protocol implementations *crosscutting* the classes into which the system has been decomposed. As a result, the software engineering properties of systems—such as traceability, comprehensibility, and maintainability—tend to suffer. Errors can be introduced when an implementation must be produced that necessarily bears little resemblance to the specification. Corrective maintenance is difficult when the connection between the specification and the implementation is unclear and the implementation itself is complex. Although some protocols are targets for standardization, standards themselves change over time or are replaced, as exemplified by changes to FTP, HTTP, and CORBA. Thus, protocol implementations will need to evolve either because they are non-standard or because the standards to which they adhere have evolved. Evolution is difficult and error-prone when the maintainer fails to understand the connection between the source they are changing and the original protocol.

Aspect-oriented programming (AOP) promotes the separation and modularization of the crosscutting concerns in a system. To form a functioning system, the separated concerns and the base functionality of the system must still interact. This interaction can be specified by describing a set of *join points* in the base functionality (either static points in the source code or points in the execution) at which to add or replace behaviour. In most existing concrete approaches to AOP, join point descriptions refer to properties of individual join points in isolation, without reference to other join points. Through descriptions of isolated join points, it is possible to manually translate complex communication protocols into implementations. However, such translation is less than satisfactory: a haze of details surrounding the manual implementation of protocols can obscure the original intent of even simple protocols. As a result, such AOP approaches can separate and modularize a crosscutting protocol concern but the resulting implementation remains poorly traceable, incomprehensible, and/or difficult to modify.

Since protocols generally involve sequences of events, the idea that one can describe protocols via specifications of their legal traces seems straightforward (e.g., [5, 7, 8, 11, 13]). However, all existing work along these lines falls into at least one of four categories: (1) specification techniques without a means for direct implementation [5, 14]; (2) verification and monitoring techniques that can announce events or problems but not effect the correct, resultant behaviour [7, 15]; (3) purely theoretical work [8, 11]; and (4) implementation techniques that fail to provide the software engineering properties that we need [13, 9]. Most implementation techniques require the developer to manually translate a description of the protocol into (a) instrumentation to announce the occurrence of events within the base functionality, (b) the set of points

SIGSOFT '04/FSE-12, Oct. 31–Nov. 6, 2004, Newport Beach, CA, USA.

within the base functionality at which to insert the instrumentation, and (c) an event parser for identifying when the traces of interest occur. While some work has examined the possibility of higher-level specification of trace parsing [7, 18], those approaches are not sufficiently general and are difficult to create in practice.

In this paper, we introduce *declarative event patterns* (DEPs) as a practical means to implement protocols while improving their traceability, comprehensibility, and maintainability. DEPs are descriptions of sequences of events in the execution of a system, including properly-nested event structures. DEPs allow a developer to describe a protocol in a fashion that is succinct, without the need to express extraneous details. A developer can indicate that specific actions be taken when a given pattern occurs. DEPs are compiled into the appropriate instrumentation and event parsers for recognizing a given pattern. As a result, modifying a protocol implementation requires that its DEP description be modified and re-compiled, rather than directly modifying the scattered and tangled implementation of event instrumentation and event parser.

We begin with a skeletal example of a protocol to motivate the problem, in Section 2. We have implemented a proof-of-concept tool, described in Section 3, for the use of DEPs that extends the AspectJ language [19] with advanced compiler technology [4, 3]. In Section 4, we examine a case study involving the implementation of user authentication within an FTP server. This implementation is performed via four approaches, for comparison: Java, AspectJ, the EAOPTool [13], and DEPs. Remaining issues and related work are discussed in Sections 5 and 6, respectively.

This paper provides the following contributions. Declarative event patterns provide a mechanism for matching context-sensitive join points, beyond those provided by AspectJ or other AOP approaches. We describe a proof-of-concept implementation of DEPs. We demonstrate that DEPs can improve the traceability, comprehensibility, and evolvability of protocol implementations.

2. MOTIVATION

To motivate the problems encountered in recognizing event patterns, consider a skeletal example of a protocol that is to be implemented in a system that includes four methods of interest: `basic`, `complex`, `safe`, and `unsafe`.

The behaviour of `complex` must differ depending on the state of the system, according to the following two constraints. (1) If `basic` has been called, it will have established properties that `complex` must use. If `basic` has not been called, `complex` is free to define properties that are more tailored for its context. For example, `basic` may already have established a connection with a remote service that was easy to locate but that provides relatively poor quality of service; this service suffices for the fulfillment of `basic`. On the other hand, it might be preferable to spend more resources locating a different remote service with better quality of service for the sake of `complex` if a connection does not already exist. (2) The `safe` and `unsafe` methods can be called in a mutually recursive structure, and the control flow may proceed from either to call `complex`. If a call to `safe` encloses a call to `complex` more closely than a call to `unsafe`, then `complex` can perform its behaviour in an efficient manner that assumes safety properties. Otherwise, a less efficient mechanism will be used. For example, `safe` and `unsafe` may involve locking and unlocking resources in a transactional manner.

A possible execution tree through such a system is depicted in Figure 1. The boxed event highlights the only call to `complex` that is safe but constrained by the properties previously established by `basic`. Only enough of the tree is shown to illustrate some of its structure of interest; a real tree would contain many more

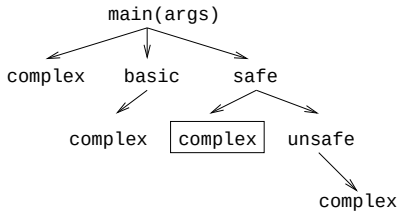

Figure 1: A partial execution tree involving a protocol.

method calls and other events (such as field accesses, class loads, or object creation). The sequence of events in this system can be identified through a pre-order traversal of the tree. Such a sequence is called a *trace*.

To react to the patterns of significance in the execution tree, we can identify the patterns in two stages: *lexing* individual events and *parsing* the resulting lexemes to recognize patterns of multiple events. There are two issues involved in such identification: implementation of the means of recognition, and specification of the patterns of interest.

To recognize patterns of multiple events, an automaton will ultimately be needed at run-time. This is true whether a developer manually implements the automaton directly or via a high-level specification, and whether the implementation details of the automaton state and state transitions are spread across the system or are well-modularized. In many circumstances, a finite state machine would suffice to recognize these patterns but not in the execution tree shown. For example, the properly nested entries and exits of `safe` and `unsafe` must be recognized to determine when `complex` can utilize an efficient but dangerous algorithm. Recognition of these properly nested events is reducible to recognition of properly nested parentheses; hence, a finite state machine will not suffice in general, but an automaton such as a pushdown will be needed.

To specify an execution pattern of interest, the developer must determine the pattern of interest and implement the automaton for recognizing it. A direct, manual implementation of the protocol is likely to suffer from the typical shortcomings of crosscutting concerns: a lack of traceability, comprehensibility, or evolvability—the software engineering properties of interest in this paper. Separating and modularizing the protocol implementation should improve the situation; however, the implementation will still be low-level, difficult to connect to the requirements, and difficult to modify for the sake of debugging or evolution. There are two well-tested, high-level mechanisms for specifying patterns of interest: regular expressions (e.g., in the Unix `grep` tool) and context-free grammars (e.g., as used by parser generators). Since regular expressions are not capable of expressing patterns with proper nesting, like that of our example protocol, we go the route of more general context-free grammars (CFGs). Fortunately, a CFG specification can be kept simple when a regular expression will suffice. And in generating the pushdown automaton realizing the CFG, a given concrete system can be statically analyzed to limit the use of the pushdown whenever possible, thereby eliminating much run-time overhead. Previous work describes static analyses that can be performed automatically (e.g., [4, 24]). We now consider how these ideas can be realized in practice.

3. DECLARATIVE EVENT PATTERNS: A PROOF-OF-CONCEPT TOOL

We have created a proof-of-concept tool that allows the developer to specify a protocol (or any other behaviour) as a context-free

grammar plus a set of behaviours to perform when patterns of interest occur; these descriptions are given as *declarative event patterns* (DEPs)—descriptions of multiple-event patterns.

Our proof-of-concept tool permits a developer to augment their AspectJ (version 1.1) source code with DEPs. The tool then translates this augmented AspectJ source into standard AspectJ source. This output source implements (1) an event parser to recognize each declarative event pattern, (2) the instrumentation code that will announce the occurrence of particular events at run-time to the event parsers, and (3) the specification of the join points where the instrumentation must be injected. With this approach, we need only instrument those points in the system that can generate events that affect the state of the event parsers, and need not keep a permanent record of those events.

We begin by describing the syntax and informal semantics of our proof-of-concept tool, in Section 3.1. Elements of the skeletal example protocol we examined in Section 2 are used to illustrate the use of the tool. We describe some details of implementation of the tool in Section 3.2. Remaining issues are delayed until Section 5.

3.1 Syntax and Informal Semantics

In this section, we provide a brief overview of AspectJ and proceed to our DEP extensions that build upon it. Readers unfamiliar with the syntax and semantics of AspectJ 1.1 should refer to [25] for further details.

AspectJ provides three constructs of interest in this paper: pointcuts, advice, and aspects. A *pointcut* is a specification of the set of points (called *join points*) within the execution of a program or within its static source at which behaviour is to be added or replaced. AspectJ defines a set of *primitive pointcuts* for capturing join points such as method executions, field sets, or class initializations. Primitive pointcuts may be combined through conjunction (&&) and disjunction (||) operators to build up more complex expressions. Pointcuts can be named to increase readability and to reuse their definitions. To add or replace behaviour at a set of join points, a developer declares *advice* to operate on the appropriate pointcut; advice can be declared that operates immediately before, immediately after, or in place of ("around") each join point matching a pointcut declaration. The body of each piece of advice implements the behaviour to be inserted. Pointcuts can expose state for use within the advice body, such as the actual arguments passed in a method call. The advice and pointcuts related to a particular crosscutting concern are collected within a class-like construct called an *aspect*. The AspectJ compiler "weaves" together aspects and base functionality to effect the advice behaviour as specified.

To this base, we have added two constructs: *tracecuts* and a `history` primitive pointcut. Each tracecut is a declarative event pattern specification. Tracecuts can be named. The `history` primitive pointcut takes a tracecut (or reference to a named tracecut) as an argument. Pointcuts that include the `history` primitive pointcut may be advised like any other pointcut; the advice is executed when the declarative event pattern is matched by the pattern of events in the actual execution. Figure 2 shows the syntax for our extensions. Rules beginning with a capital letter correspond to the standard AspectJ or Java interpretations.

Primitive tracecuts define the lexemes of declarative event patterns, capturing individual events in the execution trace. Two primitive tracecuts are provided by the tool. Entrance 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. To

history_pc	→	**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 2: Grammar for our proof-of-concept tool.

define constraints on the temporal or causal ordering of events, ordered tracecuts may be defined as production rules. Each ordered tracecut is an expression involving primitive tracecuts or references to named tracecuts; concatenation, Kleene closure, disjunction, recursion, and various forms of syntactic sugar are provided.

An example of a primitive tracecut is shown in Figure 3. A tracecut named `basicDEP` 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 `basic`; the argument passed to this method is exposed in the formal parameter `i`. The `basicDEP` tracecut is then used within `before` advice. The `complexPC` pointcut specifies the join points where the `before` advice should apply, as long as the method `basic` was entered at any time in the past. 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 `basicDEP` within the `history` pointcut with the explicit right-hand-side of the production rule.

```
tracecut basicDEP(int i) ::=
  entry(execution(* *.basic(int)) && args(i));
pointcut complexPC():
  execution(* *.complex(..));
before(int i):
  complexPC() && history(basicDEP(i)) {
  // Advice implementation making use of i
}
```

Figure 3: An example of a primitive tracecut.

An example of an ordered tracecut is shown in Figure 4. The `isSafe` tracecut specifies a declarative event pattern where we want to recognize a nested sequence of calls involving `safe` and `unsafe`. Specifically, we want to ensure that the current event is more tightly enclosed in an execution of `safe` than `unsafe`. Two pointcuts are declared to capture executions of `safe` and `unsafe`. The `completed` tracecut captures all completed, properly nested entry–exit pairs on `safe` or `unsafe`. The dollar sign matches the end of the execution trace that has been encountered up to the current moment in the execution. The `isSafe` tracecut will thus match the execution at a given moment only if there is an unmatched entry to `safe` and no unmatched entries to `unsafe`.

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. This

```
pointcut safePC(): execution(* *.safe(..));
pointcut unsafePC(): execution(* *.unsafe(..));

tracecut completed() ::=
  (entry(safePC()) [completed()] exit(safePC())) |
  (entry(unsafePC()) [completed()]
    exit(unsafePC()));

tracecut isSafe() ::=
  entry(safePC()) completed()* $;
```

Figure 4: Capturing properly nested sequences.

is a standard feature of parser generators.

Figure 5 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 `basic`; however, we need the named tracecut that we are declaring (namely `basicDEP`) to expose a value of type `int`. Thus, we declare a local variable `arg` of type `Integer` to capture the argument exposed from the call to `basic`, and then convert this argument to an `int` value within the semantic action block.

```
pointcut basicPC(Integer arg):
  call(* *.basic(Integer)) && args(arg);

tracecut basicDEP(int i) {: Integer arg :} ::=
  entry(basicPC(arg))
  {: i = arg.intValue();
     if(i & 1) /* is odd number? */
       fail; /* reject this occurrence */ :};
```

Figure 5: Using semantic action blocks.

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 `basicDEP` 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 indicate no semantic failure for a match.

3.2 Tool Implementation

Our proof-of-concept tool translates aspects augmented with DEPs into standard AspectJ. We consider here a few details of how the tool generates instrumentation and the event parser. Further details can be found elsewhere [26].

For each `history` primitive pointcut for which advice is applied, the tool 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` primitive pointcut. These automata are implemented as optimized, 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 tool applies token-generation advice for each parser (and hence, each DEP) 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 normally need to be ordered. We have conducted initial investigation into the use of general LR recognition coupled with the Schrödinger's token approach [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` primitive pointcut is translated into an `if` primitive pointcut 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.

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.

With primitive tracecuts, their associated semantic 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 semantic actions execute every time an occurrence of the primitive event is encountered during execution. With ordered tracecuts, semantic actions are executed immediately after the preceding tracecut pattern is recognized. An ordered tracecut may use this opportunity to manipulate state from its constituent parts. As mentioned earlier, tracecuts may also reject the occurrence of the pattern based on some semantic condition, causing the parser to return to its initial state.

4. CASE STUDY: FTP AUTHENTICATION

We now consider a case study that we have conducted involving the development and extension of a server for the File Transfer Protocol (FTP). 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 [22]) 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 6; 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.

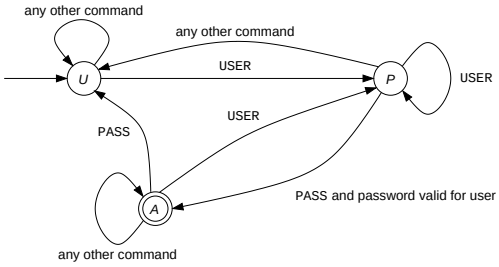

Figure 6: FSM for authentication implied by RFC 959.

Initially, we designed and implemented in Java an FTP server providing the Minimum Implementation subset as specified by RFC 959. This base design (Figure 7) ignored the presence of the remainder of the FTP specification, including the user authentication protocol. The `Server` listens at a port for connection attempts by clients, which result in the creation of a new `Session` and its associated `ControlConnection` and `TransferContext`. The `ControlConnection` is used to send commands to and receive responses from the server. The `DataConnection` is a potentially transient connection opened to transfer files. `TransferContext` maintains state regarding the session and connections, such as the IP address and the file type. Each FTP command was implemented in a separate subclass (via the Command design pattern); each is responsible for parsing its own arguments and sending its own responses. The appropriate `Command` subclass is selected by the `Interpreter` by parsing the command string received from the client. `CommandFactory` creates or caches instances of `Command` subclasses.

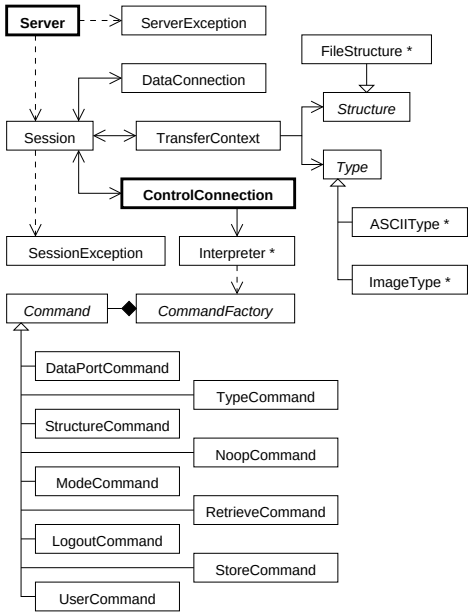

Figure 7: Base design for the FTP server.

The base design was then extended to add the user authentication feature, one version for each of four approaches: Java (Section 4.1), AspectJ (Section 4.2), EAOPTool (Section 4.3), and our proof-of-concept tool for DEPs (Section 4.4). In Section 4.5, we compare the resulting implementations for their effects on traceability, comprehensibility, and evolvability.

4.1 FTP Authentication in Java

Authentication required additions to several classes in the base design. A `PasswordCommand` was added to the `Command` hierarchy. 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 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 `PasswordCommand` class was manually instrumented to report occurrences of `PASS` events, `UserCommand` reported `USER` event occurrences, and all other `Command` subclasses reported `OTHER` event occurrences.

Figure 8 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, 6 of the 9 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`.

4.2 FTP Authentication in AspectJ

The AspectJ extension (Figure 10) followed an approach to user authentication similar to that in the Java extension. A single aspect was created, along with `AuthenticatorMonitor` and `PasswordCommand` classes equivalent to those added for the Java extension.

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 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.

4.3 FTP Authentication in Event-Based AOP

Event-based AOP (EAOP) [12, 13] 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 coroutine 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 9. 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 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 6 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.

4.4 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 11. The five pointcuts we used here are identical to those found in the original AspectJ aspect of Figure 10.

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 authentic 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.

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.

4.5 Comparison

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 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 Section 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 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. But all the techniques can make use of the Turing-equivalence of the underlying base language when pressed.

The declarative expression of event patterns is non-existent to poor for most of the approaches. Java and EAOPTool have none, requiring the use of the base language to manually implement the pattern recognition algorithms. AspectJ possesses only the cflow

App	In	Pa	DE	Tr	Un	Ev
Java	man	man	L	L	L	L
AspectJ	gen	man	L-M	M	L-M	M
EAOPTool	gen	man	L	L	L	M
FSM gen.	gen	gen	M	M	M	M-H
DEPs	gen	gen	H	H	H	H

Table 1: Comparison of different extension approaches. Table entries indicate manual vs. generated or Low, Medium, or High; categories are: means of Instrumentation; means of creating Parser; and resulting Declarative Expressibility, Traceability, Understandability, and Evolvability, specifically regarding event patterns.

```
public class AuthenticationMonitor {
  /* State and transition definitions */
  private abstract class AuthenticationState {
    public void observePasswordCommand(String pwd) {}
    public void observeUserCommand(String usr) {}
    public void observeOtherCommand() {}
  }

  AuthenticationState unauthenticatedState =
  new AuthenticationState() {
    public void observeUserCommand(String usr) {
      state = pendingState;
      userName = usr;
    }
  };

  AuthenticationState authenticatedState =
  new AuthenticationState() {
    public void observePasswordCommand(String pwd) {
      state = unauthenticatedState;
      userName = null;
    }
    public void observeUserCommand(String usr) {
      state = pendingState;
      userName = usr;
    }
  };

  AuthenticationState pendingState =
  new AuthenticationState() {
    public void observePasswordCommand(String pwd) {
      if(Authenticator.isValid(userName, pwd)) {
        state = authenticatedState;
      } else {
        state = unauthenticatedState;
      }
    }
    public void observeOtherCommand() {
      state = unauthenticatedState;
    }
    public void observeUserCommand(String usr) {
      userName = usr;
    }
  };

  /* Protocol context */
  private AuthenticationState state =
    unauthenticatedState;
  private String userName = null;

  /* Mediation method */
  public boolean isAuthenticated() {
    return (state == authenticatedState);
  }

  /* Observation methods */
  public void observeOtherCommand() {
    state.observeOtherCommand();
  }
  public void observeUserCommand(String usr) {
    state.observeUserCommand(usr);
  }
  public void observePasswordCommand(String pwd) {
    state.observePasswordCommand(pwd);
  }
}

/* Extensive modifications to base functionality
   are necessary but not shown... */
```

Figure 8: Partial authentication extension in Java.

pointcut and its closely related variants; we discuss it further in Section 5. For FSM-based generators, properly nested events cannot be recognized. DEPs possess the highest degree of declarative expressibility of event patterns amongst the approaches investigated.

The traceability of most of the AOP approaches is improved by the separation of the authentication protocol from the base functionality. However, the traceability of the event pattern remains poor in all the approaches except DEPs. Traceability of these pat-

```
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 pointcut definitions

// Action language specification not shown
// but conceptually equivalent to the
// AspectJ behaviour modification advice
```

Figure 9: Partial authentication extension in EAOP.

```
public aspect AuthenticationMonitor {
  /* State and transition definitions, protocol
     context, and mediation method all identical
     to Java extension, so not repeated... */

  /* Observation pointcuts */
  pointcut user(String usr):
    execution(*
      UserCommand.perform(String, Session))
    && args(usr);

  pointcut pass(String pwd):
    execution(*
      PasswordCommand.perform(String, Session))
    && args(pwd);

  pointcut other():
    execution(* Command+.perform(..))
    && !user(*) && !pass(*);

  pointcut quit():
    execution(*
      QuitCommand.perform(String, Session));

  pointcut needAuthentication(Session s):
     other() && !quit() && args(s);

  /* Behaviour modification */
  void around(Reply r):
    cflow(user(String))
    && args(r)
    && call(* * ControlConnection.send(Reply)) {
      proceed(new NeedPasswordReply());
  }

  void around(Session s): needAuthentication(s) {
    if(!isAuthenticated()) {
      ControlConnection cc =  s.getControlConnection();
      cc.send(new NotLoggedInReply());
    } else {
      proceed(s);
    }
  }

  void around(Reply r):
    cflow(pass(*)) && args(r) &&
    call(* * ControlConnection.send(Reply)) {
    if(!isAuthenticated()) {
      proceed(new LogInFailedReply());
    } else {
      proceed(r);
    }
  }

  /* Observation instrumentation */
  before(String usr): user(*) && args(usr) {
    state.observeUserCommand(usr);
  }
  before(String pwd): pass(*) && args(pwd) {
    state.observePasswordCommand(pwd);
  }
  before(): other() && !quit() {
    state.observeOtherCommand();
  }
}
```

Figure 10: Partial authentication extension in AspectJ.

terns suffers because they must be explicitly and manually translated into a set of states 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 EAOP version and the `userName` and `state` fields in the AspectJ version) or the explicit states that had to be identified in the finite state machine represented in Figure 6. The developer can see

```
public aspect AuthenticationMonitor {
  /* Observation pointcuts */
  pointcut user(String usr):
    execution(*
      UserCommand.perform(String, Session))
    && args(usr);

  pointcut pass(String pwd):
    execution(*
      PasswordCommand.perform(String, Session))
    && args(pwd);

  pointcut other():
    execution(* Command+.perform(String, Session))
    && !user(*) && !pass(*);

  pointcut quit():
    execution(*
      QuitCommand.perform(String, Session));

  pointcut needAuthentication(Session s):
     other() && !quit() && args(s);

  /* Event pattern detection */
  tracecut isAuthenticated()
    {: String name, String pwd :} ::=
      entry(user(name)) entry(pass(pwd))
      {:  if(!Authenticator.isValid(name, pwd))
            fail;
      :}
      entry(other())* $;

  /* Behaviour modification */
  void around(Session s):
    needAuthentication(s)
    && !history(isAuthenticated()) {
     ControlConnection cc = s.getControlConnection();
     control.send(new NotLoggedInReply());
  }

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

Figure 11: Full authentication extension using DEPs.

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, i.e., evolvability is coupled with comprehensibility. Our own experience with defining finite state machines from informal specifications, as well as the experience of others [30], supports this contention.

We were able to represent the regular expression for user authentication directly with the use of declarative event patterns as provided by our proof-of-concept tool—no translation to other 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. DISCUSSION

In this section, we consider a number of remaining issues surrounding the concept of declarative event patterns and our proof-of-concept implementation.

Pointcuts versus tracecuts. Rather than provide DEPs 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. While all AOP approaches operate on the basis of a join point model, join point models differ between different approaches. In AspectJ, dynamic join points are effectively points in the control-flow graph of a program. In declarative event patterns, dynamic join points are event occurrences. The fact that tracecuts and pointcuts are combined in expressing declarative event patterns brings into question the nature of the relationship between the two and whether both are necessary.

The AspectJ join point model does not easily support multi-point patterns because its join points are not well-ordered. To see this, consider again the example from Section 2 where the method `safe` is calling the method `complex`. A straightforward interpretation of this situation might be that `safe` executes before `complex`. However, consider applying `before` and `after` advice to the `execution` of each of these methods, which should execute respectively immediately before and immediately after each of the methods. The `before` advice on `safe` would execute before that on `complex`, but the `after` advice would execute in the reversed order. In other words, the `execution` pointcut does not describe a discrete event but an interval, from a trace-based perspective.

This difference does not matter much when join points are considered in isolation. In multi-point patterns, we can see 3 possible solutions. (1) A more elegant realization of DEPs would replace AspectJ's join point model in favour of one based on discrete events. This would require a significantly different language from AspectJ, thereby eliminating the benefit gained from the efforts at developing AspectJ as an industrial strength approach. (2) Eliminating the discrete event model of DEPs in favour of the control-flow graph-based model of AspectJ would likely require support for join point patterns based on some interval logic. It is unclear whether such an approach would maintain the intentionality provided by DEPs. (3) Pointcuts and tracecuts can both be supported, as we have done. This solution requires that they be conjoined in a tightly controlled and slightly clumsy fashion, i.e., isolation of tracecuts inside the `history` primitive pointcut, and isolation of pointcuts inside the `entry` and `exit` primitive tracecuts. Solution 3 struck the best balance for our research purposes, but may not be the best solution in the long term.

The `cflow` and related pointcuts. AspectJ provides the `cflow(`*pc*`)` pointcut to detect join points that occur while the execution remains within the control flow of another pointcut (*pc*). For example, the pointcut

```
cflow(execution(* *.safe()))
```

would capture all events that occurred while `safe` remained on the call stack. Note that recognizing the `execution` pointcut does not suffice to recognize the `cflow` pointcut; for the latter, the beginning of the execution must be recognized and the absence of the ending of the *same* execution must also be recognized. Thus, recognizing this pattern requires that nesting of method entry and exit events be accounted for (otherwise, the first exit from a deeply recursive execution of `safe` would be interpreted as exiting the control flow of the outermost `safe`).

The difference between `cflow` and other pointcuts is significant. Since the `cflow` pointcut must recognize properly nested method entries and method exits, it must recognize a context-free language that is non-regular. Hence, its implementation requires the use of a stack. (Static optimizations can sometimes reduce the use of the stack [4, 24].) Nevertheless, `cflow` can only be used to express limited context-free languages—e.g., note that, for example, it cannot distinguish between the call sequences `safe`→`unsafe`→`complex` and `unsafe`→`safe`→`complex`, which would be problematic in the skeletal example protocol described in Section 2.

One should realize that the `cflow` pointcut is not necessary in an absolute sense. One could implement `cflow` by individually advising the entries and exits and manually implementing a stack. However, such a manual implementation would cause the purpose of `cflow` to disappear behind a haze of details scattered amongst advice and automata. The `cflow` pointcut is a higher-level specification. Declarative event patterns continue this trend towards high-level specification, thereby improving comprehensibility, traceability, and evolvability.

Avoiding further language extensions. DEPs are general purpose, since they can express arbitrary context-free patterns of events. Without them, one could encode specific patterns in new primitive pointcuts and extend AspectJ with these additions. However, DEPs allow the expression of other multi-point patterns without requiring further language extensions for each new pattern of interest.

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 traces. The `cflow` pointcut is a hard-coded example of what a generic pattern could define. Its definition would be similar to the example shown in Figure 4.

Closed universe of tokens. In our example tracecuts in Section 3.1, the `isSafe` and `completed` tracecuts are assumed to operate on a closed universe of events that consist only of entries and exits to the `safe` and `unsafe` methods. Only the primitive tracecuts in the transitive closure of the target tracecut will send tokens to the corresponding event-parsing automaton.

This approach has advantages and disadvantages. On the one hand, this allows us to simplify the expression of DEPs: 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 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.

A means is needed to specify the universe of tokens to be considered by an automaton when the universe of tokens 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 extensions are currently unrealized but straightforward.

Run-time efficiency. The space requirements of the approach depend on two things: (1) the number of points in the system at which event-generating instrumentation must be inserted, and (2) on the amount of data that must be stored at run-time to encode

the state of the pattern recognition automata. The LR parser technology that we employ limits unnecessary use of the stack thereby improving efficiency, but at the cost of a larger automaton memory footprint.

We are currently implementing additional optimizations based on static detection of definite or infeasible paths, at which time we will collect benchmark data on space and efficiency.

Multi-threading. Our proof-of-concept tool currently allows for DEPs to be matched within a single thread; its implementation creates automata strictly on a per-thread basis. The tool's implementation can be extended to deal with certain limited cases of multi-threading, by creating automata on a per-virtual-machine basis. The use of declarative event patterns 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 is likely possible [21], but remains non-trivial future work. In practice, we have rarely needed multi-threaded queries.

Dynamic definition of event patterns. The dynamic introduction of code could pose difficulties for our technique in some situations. If the new code contained DEPs that required access to details of the current trace prefix that were not being stored by an automaton already present, 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.

Formal specifications and logic-based implementations. We have used the State design pattern in implementing some of the versions of the FTP server; however, the point would have remained the same had a different means been used to implement the finite state machine (FSM) representation of the user authentication protocol. The FSM representation caused difficulties because it needed to be derived from a few informal descriptions within the FTP specification, and this derivation process was difficult and error-prone. One might argue that the FTP specification should have been more formal in the first place, thereby mitigating this difficulty. However, someone would have still needed to have generated the FSM in order for it to be present in the specification. Such derivation must necessarily, ultimately start from an informal base. Yellin and Strom have reported similar difficulties in generating state machine representations of translation protocols for type adaptation [30].

Similarly, one could ask whether a temporal or modal logic would be more appropriate for the expression of event patterns [1]. Our initial attempts at providing an event pattern language were based in temporal logics. The result of expressing a simple pattern in CTL was a week's effort to produce a whiteboard full of symbols without apparent connection to the original pattern; the equivalent pattern expressed as a DEP required 30 seconds of effort and was rather obviously the same as the concept being represented. Others have noted that "fixpoint logics are notorious for being incomprehensible" [6].

In some situations, logic-based event languages might be more intentional than DEPs. Neither an approach based on context-free grammars nor a logic-based approach is likely to be ideal in all circumstances for all purposes. As with so many things in software engineering, one size is unlikely to fit all.

6. RELATED WORK

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 [5]; or run-time verification and monitoring techniques, such as intrusion detection [27] and event-triggered decision support systems [9]. Colcombet and Fradet [8] 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 [23]. Declarative event patterns use similar techniques but specifically for the purposes of high-level implementation.

The work of De Pauw *et al.* [10] attempts to automatically discover emergent patterns in executions, rather than specify behaviour that should execute when expected patterns occur.

Feather and colleagues have investigated a variety of techniques related to DEPs. Gist is a specification language that permits the use of historical references [14]; however, Gist is not automatically compilable and thus is insufficient as a means of implementation.[1] Fickas and Feather have investigated requirements-based, run-time monitoring of programs in dynamic environments [15]. Their work has culminated in the Formal Language for Expressing Assumptions (FLEA) and its inclusion in TriggerWare, a commercial infrastructure for decision support systems [9]. TriggerWare supports run-time monitoring of instrumented programs and the recording of complete execution traces; it requires manual intervention to evolve the configuration of these systems. Despite its otherwise rich language for describing complex events, FLEA does not support the detection of properly nested event structures but is limited to sequencing. In contrast, DEPs do not require complete execution traces to be stored persistently; DEPs can express properly nested event structures; and DEPs are used within a program's implementation, not necessarily for external monitoring.

Filman, Havelund, and their colleagues have used largely manual instrumentation and event parsing techniques, particularly for verification (e.g., [16]). Douence *et al.* have advocated an event-based approach to aspect-oriented programming [12], realized in the EAOPTool [13] that we have discussed. Douence *et al.* have also developed a theoretical framework for "stateful aspects" [11], which can be used for analysis of event patterns.

A number of techniques force the developer to explicitly identify states and state transitions. Type adaptation [30] allows modules to be composed via stateful translation protocols specified as finite state machines; 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 [17]. Butkevich *et al.* [7] use explicit state-and-transition representations of FSMs to aid in debugging object protocols. JAsCo [18], a tool that combines AOP and components, claims to provide stateful aspects as described by Douence *et al.*; however, the FSM states and transitions in a protocol must be explicitly described. As discussed in Section 4.5, 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 [30] agree, indicating that it is diffi-

[1] We have been unable to locate more recent work on Gist or details on the form of these historical references. Presumably, Feather's more recent work on FLEA represents the direction taken by the research surrounding Gist.

cult to specify these state machines. Similarly, we had difficulty in correctly defining the FSM for FTP authentication (Figure 6), despite reviews by multiple colleagues. Testing and elimination of other possible error sources was necessary prior to discovering the incorrect specification.

Declarative event patterns are a practical realization of *call history*, which we have previously introduced [29, 28]. The original implementation of call history recorded every event in a system for the duration of an execution. Events could be retrieved for examination using only a simplistic application programming interface; pattern recognition was limited to basic ordering relations, such as finding the most recent occurrence of an event. DEPs address these shortcomings.

7. CONCLUSIONS

We have presented declarative event patterns (DEPs) to permit the intentional specification of patterns of multiple events. We have created a proof-of-concept tool for the specification of DEPs based on context-free grammars. This tool augments AspectJ aspects with DEP constructs that can be advised similarly to pointcuts. The tool transforms the DEP constructs into standard AspectJ pointcuts and advice that specifies event-generating instrumentation and an event-recognition parser. This parser takes advantage of advanced compiler technology. Additional static optimization of the parser is possible but remains to be added to the proof-of-concept tool.

While formal specifications and formal implementations have an important role to play in software development, they tend to be too expensive to apply in everyday situations. DEPs promise to reduce the gap between informal specifications and implementation. By providing a means of directly expressing the patterns present in informal specifications, we reduce the likelihood of losing sight of the original requirements.

We have compared AspectJ extended with DEPs to Java, event-based AOP, and standard AspectJ in a small case study involving the implementation of user authentication in an FTP server. DEPs allowed for the implementation of user authentication in a manner that improved the comprehensibility, traceability, and evolvability of the system above the other approaches.

8. ACKNOWLEDGMENTS

We thank Andrew Eisenberg, Martin Robillard and the anonymous reviewers for their comments, and John Aycock for his help with compiler technology. This work was funded in part by an NSERC Discovery Grant and by a University of Calgary Research Grant.

9. REFERENCES

[1] R. Åberg et al. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Int'l Conf. Automated Softw. Eng.*, 2003.

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

[3] J. Aycock and N. Horspool. Schrodinger's token. *Software—Practice & Experience*, 31(8), 2001.

[4] J. Aycock, N. Horspool, J. Janouek, and B. Melichar. Even faster generalized LR parsing. *Acta Informatica*, 37(9), 2001.

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

[6] J. Bradfield and C. Stirling. Modal logics and mu-calculi. In *Handbook of Process Algebra*. Elsevier, 2001.

[7] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and tool support for debugging object protocols. In *Int'l Symp. Foundations Softw. Eng.*, 2000.

[8] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Symp. Princ. Progr. Lang.*, 2000.

[9] Cs3 Inc. *TriggerWare: Infrastructure for Event Reasoning Applications*, Version 1.0, 2004. `www.cs3-inc.com`.

[10] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *USENIX Conf. Object-Oriented Tech. and Sys.*, 1998.

[11] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Int'l Conf. Generic Prog. and Component Eng.*, 2002. LNCS 2487.

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

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

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

[15] S. Fickas and M. Feather. Requirements monitoring in dynamic environments. In *IEEE Symp. Requirements Eng.*, 1995.

[16] R. Filman and K. Havelund. Source-code instrumentation and quantification of events. In *Wkshp. Foundations Aspect-Oriented Lang. at AOSD*, 2002.

[17] D. Hoffman and P. Strooper. State abstraction and modular software development. In *Int'l Symp. Foundations Softw. Eng.*, 1995.

[18] JAsCo tool. `ssel.vub.ac.be/jasco`, 2004.

[19] G. Kiczales et al. An overview of AspectJ. In *Proc. European Conf. Object-Oriented Progr.*, 2001. LNCS 2072.

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

[21] E. Ochmański. Recognizable trace languages. In *The Book of Traces*. World Scientific, 1995.

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

[23] S. Reiss and M. Renieris. Encoding program executions. In *Int'l Conf. Softw. Eng.*, 2001.

[24] D. Sereni and O. de Moor. Static analysis of aspects. In *Int'l Conf. Aspect-Oriented Softw. Dev.*, 2003.

[25] The AspectJ Team. *The AspectJ Programming Guide*. Palo Alto Research Center, Inc., 2003.

[26] K. Viggers and R. Walker. An implementation of declarative event patterns. TR 2004-745-10, Univ. of Calgary, 2004.

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

[28] R. Walker. IConJ 0.1: A proof-of-concept tool for the application of the implicit context model to Java software. TR 2004-757-22, Univ. of Calgary, 2004.

[29] R. Walker and G. Murphy. Implicit context: Easing software evolution and reuse. In *Int'l Symp. Foundations Softw. Eng.*, 2000.

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