An Implementation of Declarative Event Patterns

Technical Report
2004-745-10

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

December 8, 2004

Abstract

Characterizing patterns of events that occur during the execution of a software system and how the program should respond to such patterns is an important and a natural way to think about stateful crosscutting concerns. Our work on *declarative event patterns* (DEPs) has lead to the development of a language that allows for patterns in a program's execution to be expressed as context-free patterns of events, and for the occurrences of these patterns to alter the course of the program's execution. As a companion to a conference paper introducing DEPs, this technical report covers an initial realization of our declarative event pattern language that leverages the power and applicability of aspect-oriented programming (AOP). We have added to AspectJ (a popular Java implementation of AOP) two straight forward language constructs to support the recognition of patterns of events. Our proof-of-concept implementation takes programs implemented in AspectJ augmented with our DEP constructs and translates them into programs implemented in standard AspectJ, equipped to recognize and respond to patterns of events as they occur in the execution of the system.

1 Introduction

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 or points in the execution) at which to add or replace behaviour. Most aspect-oriented approaches support the description of individual join points in isolation. A large class of concerns cut across the base functionality of a system and involve particular patterns of events—events that may occur at dispersed points within the execution

of the base behaviours. While it is certainly possible to recognize such patterns by using hand-coded approaches involving isolated join points, this approach is both awkward and inflexible to change. The original intent of the pattern tends to become obfuscated, making it prone to error and difficult to understand and evolve.

Declarative event patterns (DEPs) free the developer from the low-level details of recognizing patterns in the execution of the program. Through a higher-level declarative syntax one can state patterns of events that may occur in the execution of the system and specify additional or alternate behaviours that are to result should the specified pattern actually occur. The low-level details of observing individual events and recognizing patterns among them is taken care of automatically through the tool support described in this report. A change to the pattern can be realized by changing the declared pattern and re-compiling. The manual alternative would involve potentially complex and error-prone low-level changes, scattered across the system.

This technical report is a companion to a conference paper that details the conceptual justifications for DEPs [15]. Here we detail the syntax, usage, and construction of a concrete realization of DEPs as an extension to AspectJ.

DEPs leverage information of past events in a given execution of a system to conditionally affect its future course. This is the idea of communication history previously introduced [14, 13], We give an overview of communication history in Section 2 and discuss some of the challenges and limitations that any realization of this idea is likely to face. DEPs are the concern of Section 3, where we establish their connection with AspectJ and introduce two new syntactic constructs and their usage as DEP extensions to AspectJ. Section 4 discusses *URD*, our proof-of-concept tool for translating programs written in DEP-augmented AspectJ into programs written in standard AspectJ. The basic transformations performed by our tool as well as some details of the parsers it constructs for the recognition of patterns of events are covered.

2 Communication History

Conceptually, the communication history of an executing program is a complete account of all communication events that have occurred since the beginning of execution up until the current point of execution. Communication events can include, for example, the entry and exit from method calls and executions, the accessing and setting of fields, object creations, etc. Importantly, the context surrounding these events (such as the arguments passed to the method call, the target object, the calling object, and return values) are also retained in this history. The benefits of providing program developers with access to such a complete and persistent history of all events that have transpired up until the point of inspection are twofold. First, a developer is able to access the state surrounding events occurring in previous execution situations for use at the present point in execution, and in doing so, bypass traditional programming constraints on the accessibility of such state. Second, a developer is able to observe meaningful patterns of events and conditionally alter the behaviour of a program should those patterns of interest occur.

One way to make the communication history of a program available to developers is shown in Section 2.1. This “strawman” implementation highlights some of the im-

mediate issues surrounding any realization of communication history. In doing so, it motivates the need for a better solution.

2.1 Strawman Implementation

With the "strawman" approach to communication history, a literally complete and persistent communication record is maintained for the duration of execution. This history is the result of source-code instrumentation, performed at compile-time, which captures and records every method entry and exit. This historic record is implemented on a per-thread basis, with all method `Call` occurrences being indexed with tree structures that maintain the causal relationship between calls, and with list structures that maintain the temporal ordering relationship between calls. Walker [13] presents a proof-of-concept tool with support for communication history; this support is provided programmatically through a collection of methods on two special runtime classes, called `History` and `Call`. The methods on the `History` and `Call` classes form an API which the developer uses to *query* the communication record for relevant events in the execution of the system. This approach allows a developer to query an ever growing history of call information. One or more `Call` objects can be retrieved from the communication history through successive calls to the API provided by the `History` class methods. These `Call`s and the information from their contexts can then be used directly at the point of inspection, as arguments for further refinement of `History` queries, or as historic reference points in the execution from which to navigate to temporally or causally neighbouring `Calls`.

The drawbacks to the "strawman" approach are self-evident. The size of the communication record grows linearly with the number of method calls in a program's execution. This growth impacts the execution speeds of the base program as the time required to perform queries on the history also grows, and demands ever more memory to represent it. In this approach the developer encodes their communication history queries programmatically through the `History` and `Call` classes discussed above. Even though history queries tend to be small, relatively simple queries expressed this way can be hard to comprehend. The strawman approach to communication history, while sufficient for initial research into the ideas of implicit context, would be impracticable in general. A better solution was needed.

3 Declarative Event Patterns

DEPs are high-level language abstractions that allow one to state in a concise declarative manner patterns in a program's execution. Like the strawman approach, these patterns involve relating the temporal and causal occurrences of communication events in a meaningful way. However, instead of programmatically making queries over an ever growing history of events, as it was with the strawman approach, DEPs declare fixed patterns over communication events. These patterns hold meaning for the developer and may conditionally influence future program behaviour should they actually be observed in the execution of the program. This fundamental shift in view from making

queries to stating patterns for observation provides avenues for efficient implementation (see Section 4).

We begin with an overview of AspectJ [6, 5], an existing AOP language extension to Java. Our initial realization of DEPs builds off of and extends AspectJ; our extensions include constructs that make it possible to state patterns of events in the execution of the system. In Section 3.2, we describe these extensions, how they are used, and how they relate to standard AspectJ.

3.1 Standard AspectJ

AspectJ considers a system to be comprised of a set of core concerns and a set of crosscutting concerns. Core concerns can be adequately represented in the base language (i.e., Java in the case of AspectJ). Crosscutting concerns by their very nature cut across the behavioural boundaries of the core concerns. Although a crosscutting concern (such as distribution, or persistence) can be represented in the base language alone, the resulting implementation of that concern becomes scattered across different parts of the system; this code is necessarily tangled amongst code implementing other concerns of the system. This makes it difficult to trace, to comprehend and to evolve not only the crosscutting concern but those core concerns in which it is tangled. AspectJ permits the separation and modularization of crosscutting concerns in an effort to regain these software engineering properties[1]. In this section we look at those fundamental features of AspectJ relevant to our DEP extensions: *aspects, advice, pointcuts,* and *context exposure*.

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

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

In Figure 1 an aspect has been declared that represents a simple crosscutting concern. This concern involves logging of entries to all methods named `perform()` that are members of `Simple` or any of `Simple`'s subclasses. The figure contains a single Aspect called `Logging`. Within this aspect one pointcut and one piece of advice are declared. The pointcut is named `loggedMethods()`. It describes the set of join points involving the execution of a method called `perform()` on the class `Simple` or any of its subclasses; the `perform()` method must additionally have a result type of `void` and have no formal parameters. The advice is declared to act on

[1]For a more complete treatment of AOP and AspectJ, the interested reader should consult elsewhere [6, 5]

the `loggedMethods()` pointcut. The advice behaviour is declared to explicitly run just before the method executions to log.

```
public aspect Logging {
  pointcut loggedMethods(): execution(void  Simple+.perform());
  before(): loggedMethods() {
   // Log the occurrence of this method entry
  }
}
```

Figure 1: An AspectJ aspect involving the logging of entries to all methods named `perform()` that are members of `Simple` or any of `Simple`'s subclasses.

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

AspectJ's pointcut language provides a number of *wildcard* features to express a wide range of joinpoints in a concise manner. The `+` symbol used in the logging example above is one such wildcard—it allows one to refer to a type or any of its subtypes in just a few characters. An ellipsis wildcard (`..`) can be used to under-specify Java packages and method parameter signatures. Kleene closure (`*`) can be used to partially specify Java identifiers. Complete coverage of AspectJ's pointcut language is available elsewhere [6, 5].

Context exposure is a feature of AspectJ that allows one to retrieve and manipulate the state surrounding a join point. Pointcuts may be declared in such a way as to expose this state for use in advice. AspectJ defines primitive pointcuts that can be used to expose the arguments (`args`), target object (`target`) and current object (`this`) with which a join point is associated. Additional reflective information about the current join point is accessible through a special variable called `thisJoinPoint` (and its variants).

The code listing in Figure 2 demonstrates an aspect exposing an integer argument so that it can be used in around advice which ensures that the argument is of a valid form before proceeding with the advised method call. The example contains a single Aspect called `ArgumentValidation`. Within this aspect one pointcut and one piece of advice are declared. The pointcut is named `callsToValidate` and has a single formal parameter `argument` of type `int`. This pointcut captures the set of of join points involving calls to method `doIt()` which have as parameter a single integer, and which are defined on the class `Element`; `doIt()` must have a result type of `void`. Additionally, this pointcut uses the `args` primitive pointcut to exposure the first (and only) argument to matching calls to `doIt()`; this argument is bound to the pointcuts formal parameter `argument`. The advice is declared to act on the `callsToValidate()` pointcut, which makes the bound argument available through its parameter. The ad-

vice behaviour is declared explicitly to run in place of (`around`) the matching method calls; if the exposed argument is negative, it is assigned the value 0 before proceeding with the matched call joinpoint.

```
public aspect ArgumentValidation {
pointcut callsToValidate(int argument):
call(void Element.doIt(int)) && args(argument);
void around(int a): callsToValidate(a) {
 if(a < 0)  a = 0;
  proceed(a);
 }
}
```

Figure 2: A demonstration of AspectJ's context exposure feature. Here a pointcut is used to expose an integer argument so that it can be used in around advice to ensure that the argument is of a valid form before proceeding with the method call.

We now consider how AspectJ can be extended to support DEPs.

3.2 AspectJ extensions for DEPS

DEPs give the developer higher-level abstractions for specifying patterns of events. With these abstractions, AspectJ advice may be conditionally applied to a join point based both on the classification of the join point (as is the case with standard AspectJ) and also on the execution patterns in which it occurs. The extensions centre around the addition of two constructs: `tracecut`s and the `history` primitive pointcut. In the following subsections we consider these constructs in turn.

3.2.1 Primitive Tracecuts

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

Examples of some primitive tracecuts are shown in Figure 3. Recall that primitive tracecuts work to specify individual events in a programs execution, and are the combination of a regular AspectJ pointcut and an indication of entry or exit. This example contains a number of unconnected tracecuts highlighting the variations of primitive pointcuts possible. The example depicts an DEPs-augmented aspect named `PrimitiveTracecuts`. Within this aspect, two named tracecuts have been declared. The first of these tracecuts is named `performEntries` and demonstrates the use of the `entry` primitive tracecut. It captures all entries into executions of method `perform()`, on class `Simple` or any of its subclasses. The second tracecut is named `someExits` and highlights the use of the `exit` primitive tracecut. It captures all

```
public aspect PrimitiveTracecuts {
 tracecut performEntries(): entry ( execution(void  Simple+.perform()));
 tracecut someExits(): exit (
  execution (void  Simple+.perform()) ||
  call (Integer  ca.ucalgary.cpsc.*.*()) ||
  execution (*  *.get*(..))
 );
}
```

Figure 3: An DEPs-augmented aspect demonstrating the `entry` and `exit` primitive tracecut extensions

exits from joinpoints in the set matched by a compound pointcut involving executions of method `perform()`, on class `Simple` or any of its subclasses, calls to any methods defined with the package prefix `ca.ucalgary.cpsc` returning type Integer and having no parameters, or any execution of methods with names beginning with `get`.

In AspectJ, a pointcut may be specified in conjunction with after advice so to refine the pointcut to cases when the join points in question are exited due to an exception being thrown or due to a normal return. Primitive `exit` tracecuts allow you to capture these events in a way syntactically equivalent to AspectJ. Figure 4 shows an example of this. An aspect named ReturningThrowingExample is defined containing two named tracecuts. The first tracecut exhibits the capture of exits from calls to methods with package prefix `ca.ucalgary.cpsc` which result in the throwing of an exception of type `UnknownException`. The second tracecut captures exits from the same set of call joinpoints, but only when those calls return with an Integer value. With both tracecuts, the thrown exception (`a`) or the returned result (`i`) are exposed and made available for later use in advice.

```
public aspect ReturningThrowingExample {
 tracecut throwingExits(): exit (
  call (Integer ca.ucalgary.cpsc.*.*())) throwing (UnknownException a);
 tracecut returningExits(): exit (
  call (Integer ca.ucalgary.cpsc.*.*())) returning (Integer i);
}
```

Figure 4: An DEPs-augmented aspect demonstrating the special exit tracecut syntax for returning/throwing events.

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

3.2.2 Ordered Tracecuts

Ordered tracecuts are the foundation of DEPs providing a means to describe potentially complex execution patterns involving precedence and dominance of events. They are used to declare patterns on the temporal or causal ordering of events, and may be defined as potentially recursive patterns involving primitive tracecuts or references to named tracecuts. Concatenation, Kleene closure, disjunction, recursion, and various forms of syntactic sugar are provided. DEP patterns are limited to those expressible by context-free languages. If one considers primitive tracecuts the terminal symbols of a context-free grammar, then ordered tracecuts can be seen as the production rules of a context-free grammar.

Ordered tracecuts are used to declare the event patterns of interest to a developer. On the surface they resembles the AspectJ pointcut. Unlike a regular pointcut, which specifies the set of valid join points of interest, a tracecut specifies the set of valid *patterns of events*. The following example in Figure 5 shows the building up of an ordered tracecuts from explicit primitive tracecuts.

```
public aspect OrderedTracecutExample {
 tracecut orderedTracecut() ::=
  entry( call (Integer ca.ucalgary.cpsc.*.*()))
  entry( call (Integer Simple.doIt(..)))
}
```

Figure 5: A DEPs-augmented aspect containing a single ordered tracecut `orderedTracecut`, comprised of two primitive `entry` tracecuts in an ordered sequence.

The example defines an aspect called `OrderedTracecutExample` containing a single ordered tracecut named `orderedTracecut()` which is comprised of two primitive entry tracecuts on pointcuts of a nature we have seen in previous examples. This ordered tracecut defines a particular sequence of these two events. When the first entry occurs during program execution, followed at some point by the second, the orderedTracecut is said to match.

Ordered tracecuts can be composed from other ordered and primitive tracecuts. Tracecuts may be named (as with `orderedTracecut` above) and referred to by these names for improved clarify and comprehension.

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

Figure 6: A DEPs-augmented aspect containing a complex ordered tracecut `complex`, composed of four primitive tracecuts.

In Figure 6, a DEP-augmented aspect is declared which demonstrates a complex ordered tracecut in use. Here four primitive entry pointcuts are declared, one for each entry to the call of methods a, b, c, and d. The tracecut named `complex` declares an ordered tracecut which specifies that execution paths initiated at the beginning of execution (via the anchoring caret symbol), up to the current point in execution (via the anchoring dollar symbol), in which first an `a` event occurs, followed by zero or more `b` events, optionally followed by another `a` event, and then either a `c` event or a `d` event. This example demonstrates the use of some of the syntactic sugar provided in the DEPs language extensions. Kleene closure (`*`) is used to provide repetition of `b` events. Square brackets ([and]) are used to specify optional tracecut parts. We use disjunction (|) to specify alternatives between `c` events and `d` events. In Figure 7 we present an example of an ordered tracecut which uses recursion.

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

Figure 7: A DEPs-augmented aspect containing a complex ordered tracecut `nesting`, which recursively defines itself.

In Figure 7, an DEP-augmented aspect contains a tracecut named `nesting` which describes a pattern of events recursively. The pattern `nesting` describes is that of a self-embedding between the primitive events `a` and `b`. `nesting` may alternatively (via disjunction) simply be the event `c` or even match against the empty stream of events (the last disjunctive part is empty). Some possible steams of events that this pattern would match include the following: (ϵ), (c), ($a \longrightarrow b$), ($a \longrightarrow c \longrightarrow b$), ($a \longrightarrow a \longrightarrow b \longrightarrow b$), ($a \longrightarrow a \longrightarrow c \longrightarrow b \longrightarrow b$), and so on.[2]

3.2.3 History Primitive Pointcut

A pointcut utilizing the history primitive pointcut extension will match those join points in conjunction with itself only when the provided tracecut pattern argument is currently satisfied. In this way, advice application may be refined to certain execution contexts. Each history designator has a target tracecut; each target tracecut is recognized independently.

In Figure 8 we apply the history pointcut to advise a set of method call join points, but only when a certain execution situation has occurred. In this example before advice is applied to methods that are named beginning with `get` defined on any class, with any number of parameters, and return type `void`, but only if the `nesting` pattern (as defined in Figure 7) has been previously observed.

[2]We use the $\longrightarrow$ symbol to denote zero or more communication events in the execution that are not a part of the alphabet of events that make up the DEP pattern.

```
public aspect HistoryExample {
    pointcut methodCalls() : call(void *.get*(..));
    before(): !history(nesting()) && methodCalls() {
      System.err.println("No nesting pattern observed.");
    }
}
```

Figure 8: A brief example of the usage of the history primitive pointcut. Before advice is applied to methods that are named beginning with `get` defined on any class, with any number of parameters, and return type `void`, but only if the `nesting` pattern has been previously observed.

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

3.2.4 Context exposure, Semantic Blocks, and Failure

One of the most powerful features of AspectJ is the ability to expose, for a given join point, the state that surrounds a join point for use within advice (this is known as *context exposure*. See Section 3.1). Context exposure is available with both primitive and ordered tracecuts. In the following example (Figure 9), we define a primitive tracecut that exposes state from a pointcut argument. The pointcut in this case is named `simplePc` and the primitive tracecut that uses this pointcut as argument and source of context is named `simpleEntry`. Here the `simpleEntry` tracecut forwards on exposed context from the `simplePc` argument. The tracecut `simpleEntry` forwards the exposed integer i of pointcut `simplePc` through its formal parameter `j`.

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

Figure 9: In this code snippet, a primitive tracecut forwards on exposed context from its pointcut arguement. The tracecut `simpleEntry` forwards the exposed integer i of pointcut `simplePc` through its formal parameter `j`.

Context exposure for ordered tracecuts works in a similar fashion. An ordered tracecut such as `exposingTracecut` in Figure 10 can forward on the exposed state from its composite tracecuts and even to a piece of advice through the use of a history pointcut. This ordered tracecut exposes two integers bound to those exposed by the two constituent primitive tracecut parts. The advice declared here uses the history pointcut to expose these values to the advice body.

As a matter of convenience, in addition to capturing exposed state solely through formal parameters, DEPs provide a way to declare temporary semantic variables; state exposed through a tracecut can also be bound to these variables and manipulated in

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

Figure 10: This ordered tracecut exposes two integers bound to those exposed by the two constituent primitive tracecut parts. The advice declared here uses the history pointcut to expose these values to the advice body.

an optional *semantic block* of Java code. These semantic variables and actions can be used in combination to alter what state is bound to a tracecuts formal parameter.

For primitive tracecuts, the use of temporaries and the semantic action block typically involves modifying the state drawn from the join point before assigning it to a formal parameter. Temporary local variables are declared in a block on the left-hand-side of a named tracecut declaration as seen in Figure 11. In this example, a primitive tracecut `primitiveTracecut` exposes an Integer argument passed to a call to method `a`, but before exposing it, it ensures that the value of the argument is not null. If the exposed argument is null, a new Integer object is created and bound to formal parameter `i` of tracecut `primitiveTracecut`.

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

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

Figure 11: A primitive tracecut `primitiveTracecut` exposes an Integer argument passed to a call to method `a`, but before exposing it, it ensures that the value of the argument is not null. If the exposed argument is null, a new Integer object is created and bound to formal parameter `i` of tracecut `primitiveTracecut`.

For ordered tracecuts, temporaries may be used to hold values exposed by the parts of the tracecut, and are used in much the same way as in the primitive tracecut case. In Figure 12 an ordered tracecut `helloWorld` exposes the concatenation of two strings exposed by its tracecut parts. The tracecuts `hello` and `world` expose strings at potentially disperse points in the execution. This tracecut brings these exposed values together for further computation.

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

Figure 12: An ordered tracecut `helloWorld` exposes the concatenation of two strings exposed by its tracecut parts. The tracecuts `hello` and `world` expose Strings at potentially disperse points in the execution. This tracecut brings these exposed values together for further computation.

The semantic action block may also be used to conditionally reject the recognition of a tracecut. We can enforce a semantic constraint on the class of events selected by the primitive tracecuts. For example, in Figure 13, `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.

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

Figure 13: The entry primitive tracecut exposes the Integer argument passed to calls of simple; failure is used to reject the occurrence of this event.

This example 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 simple; however, we need the named tracecut that we are declaring (namely simpleEntry) 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 simple, and then convert this argument to an `int` value within the semantic action block.

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

Figure 14: An ordered tracecut `requestReply` imposes a semantic constraint on the message identifier of a `Request`, and of a `Reply`. If the message identifier of the `Reply` does not match that of the `Request`, `requestReply` will fail to match.

The use of failure in semantic blocks associated with ordered tracecuts indicates that the current avenue of recognition should be abandoned. The semantic action block, as with the primitive tracecut ends if `fail` is reached. Consider Figure 14; here an ordered tracecut `requestReply` imposes a semantic constraint on the message identifier of a `Request`, and of a `Reply`. If the message identifier of the `Reply` does not match that of the `Request`, `requestReply` will fail to match.

4 The *URD* Tool

In order to experiment with the ideas of DEPs, we have constructed a proof-of-concept tool called *URD*[3]. From a practical perspective, the tool operates in much the same way as a parser generator (such as yacc, bison). It translates descriptions of event patterns (tracecuts), expressed as context-free languages, into (1) the event parser to recognize the such patterns, (2) the instrumentation code that will announce the occurrence of particular events at run-time to the event parser, and (3) the specification of the points in the source code 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 parser, and need not keep a permanent record of those events. In this section we describe at a high level the nature of the transformation performed by *URD*. We also look at how the tracecut specifications are translated to efficent pattern recognizers.

4.1 High-level structure

URD translates aspects augmented with our higher-level DEP constructs into standard AspectJ code which realizes these constructs at a lower level. The resulting standard AspectJ aspect contains all the low-level details required to recognize individual events as they occur, and to recognize patterns from these events.

The tool currently operates as a command-line preprocessor. Augmented AspectJ aspects are parsed and an AST representation is constructed. The input is analysed to identify primitive tracecuts, to give names to anonymous tracecuts, to expand syntactic sugar (such as optional tracecuts and Kleene closures), and to locate history pointcuts and their tracecut targets. The outcome is an intermediate representation of the DEP-augmented aspect which is emendable to parser construction and generative transformations. The current implementation is written in Java, and makes use of the Java CUP parser generator. At this time, the tool follows the AspectJ 1.1 language specification.

In Section 4.1.1, we present an example of tracecuts in practical use. In Section 4.2 we describe the generative transformations that are conducted by the tool to produce AspectJ code to realize DEPs. How our event parsers are derived and constructed is covered in Section 4.3.

[3]Originally, *URD* was an acronym, but its derivation has been lost

4.1.1 Running Example

The example detailed in this section highlights typical usage of DEP, and provides a concrete running example to ground discussion on the workings of the *URD* tool. Consider the example shown in Figure 15. 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 a call to `ConnectionManager.getConnection()` is more tightly enclosed in an execution of `safe` than `unsafe`. If such is the case, we can substitute this call with one to `ConnectionManager.getOptimalConnection()`. 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`. Around advice is declared to replace calls to `getConnection` with calls to `getOptimalConnection` whenever `isSafe` matches. We name the entry into `safe` with a primitive tracecut `a` and the exit from `safe` with a primitive tracecut `b`. Similarly, `c` and `d` are used to indicate the entry and exit of `unsafe`.

```
public aspect RunningExample {
 tracecut isSafe() ::= a() completed()* $;
 tracecut completed() ::= a() [completed()] b()
                        | c() [completed()] d();

 tracecut a() ::=  entry(safePc());
 tracecut b() ::=  exit(safePc());
 tracecut c() ::=  entry(unsafePc());
 tracecut d() ::=  exit(unsafePc());
 pointcut safePc(): execution(* *.safe());
 pointcut unsafePc(): execution(* *.unsafe(..));

 Connection around():
    call(Connection ConnectionManager.getConnection())
    &&  history(isSafe()) {
     return ConnectionManager.getOptimalConnection();
 }
}
```

Figure 15: A demonstration of typical DEP usage. The DEP-augmented aspect depicted here ensures through the `isSafe` tracecut that the current event is more tightly nested in an execution of safe than unsafe.

4.2 Transformations performed by the Tool

URD transforms AspectJ aspects decorated with DEP specifications into corresponding AspectJ aspects as a pre-processing step. The resulting standard AspectJ aspects resemble the source *URD* aspects but most notably, all `tracecut` and `history` declarations have been transformed; the generated aspect contains, for each tracecut which is the *target* of an history designator, the following declarations:

- advice instrumentation at each joinpoint of interest,emitting event tokens relevant for each *target tracecut*
- Event parsers recognizing the patterns described by each *target tracecut*
- Supporting event token and node class declarations

In addition to these declarations, history pointcuts are transformed, making use of the above declarations to conditionally permit advice to occur. Using the DEP specification in Figure 15 as a running example, we progressively reveal various details of the translation process in the coming sections. It should be noted that our running example does not make use of content exposure. Instead, when we discuss parts of the transformations effected by context exposure, we will expand on some part of the running example to include context exposure.

4.2.1 Standard AspectJ Declarations

All non-DEP related constructs are transcribed to the new AspectJ aspect without modification. Care is taken to maintain the relative order in which these non-DEP members are declared in the source file to ensure AspectJ in-aspect weaving precedence semantics. AspectJ semantics applies advice to joinpoint in the order in which advice appears. In figure 16 the non-DEP members of the aspect (see figure 15) are reproduced without modification.

```
public aspect Example {
 pointcut safePc(): execution(* *.safe(..));
 pointcut unsafePc(): execution(* *.unsafe(..));
}
```

Figure 16: Non-DEP related constructs are transcribed directly to resulting aspect.

4.2.2 ThreadLocal Parsers

Our proof-of-concept tool, *URD*, generates aspects that implement DEPs on a per-thread basis. That is to say, events are announced and monitored separately in each thread of control; this per-thread approach is in line with AspectJ's per-thread treatment of `cflow`. Recall that each `history` designator specifies a target tracecut for which it is interested. For each target tracecut a unique identifier is assigned and a parsing automata is generated. Figure 17 shows the generated result for our running example. In our example there is a single target tracecut, notably the named tracecut specified by `history(isSafe())`. This target is assigned the parser identifier 0. We add a `ThreadLocal`[4] parser, `parser$0`, as declared and initialized. In addition an accessor method (`getParser$0()`) is added. The reader interested in the details of

[4]The class java.lang.ThreadLocal was introduced in Java 1.2. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable.

how these parsing automata are generated from the tracecut grammar specifications are encouraged to read Section 4.3.

```
     private static ThreadLocal parser$0 = new ThreadLocal() {
        protected synchronized Object initialValue() {
            return new Parser(){
                public void next(Token t){ ... }
                public boolean isAccept(){ ... }
               // numerous details omitted for the sake of brevity
            };
        }
    };
    private static Parser getParser$0() {
        return (Parser)aspectOf().parser$0.get();
    }
```

Figure 17: Parsers are defined on a per thread basis

4.2.3 History pointcuts

History pointcuts result in two distinctive processes in the transformation and generation of DEPs-equipped aspects. First of all, the target tracecuts of a history application are used as the basis for parser generation, as we have already mentioned. Secondly, history pointcuts conditionally indicate the acceptance of a tracecut to AspectJ advice. Figure 18 shows this subtle transformation. Notice that in place of the history and its tracecut argument, a conditional if pointcut is present, which checks the current state of an event parser generated to recognize the DEP specified by the target tracecut.

```
Connection around():
   call(Connection ConnectionManager.getConnection())
   &&  if(getParser$0().isAccept()) {
    return ConnectionManager.getOptimalConnection();
}
```

Figure 18: Parsers are defined on a per thread basis

4.2.4 Primitive Tracecuts

The generation process must provide instrumentation to announce the occurrences of primitive events, which in turn feed the stream of events to the parser. In addition, references to exposed context at a primitive tracecuts given joinpoint must be maintained and passed to the parser; the generation process therefore creates a number of classes which act both as symbols for interpretation by the parser as well as parcels for exposed context information. A unique integer value is assigned for each primitive tracecut reachable by any given tracecut target, and a token class is generated. Instances of this class are fed to the parser on occurrence of the associated event. A static method is generated to access the unique integer value, and instance fields of the appropriate types are added to this class for each of context object exposed. Constructors and accessory methods are generated so that instances of these tokens may be constructed

by providing the exposed state and so that exposed state may be retrieved from such instances. We modify the primitive tracecut a in this example so that it exposes an argument; Figure 19 shows this modification. In figures 20 and 21 the primitive tracecuts a and b from the running example are transformed. Figure 20 shows the transformation of the tracecut as it appears in the resulting aspect, were as Figure 21 shows the generated classes that result.

```
tracecut a(String s) ::=  entry(safePc(..) && args(s));
```

Figure 19: A modified version of primitive tracecut a so to demonstrate the transformations and generation of code the result to maintain exposed state.

```
before(String a0): safePC(..) && args(a0) {
   getParser$0().next(new Urd$a$Token(a0));
}
after(): safePC() {
   getParser$0().next(new Urd$b$Token());
}
```

Figure 20: The in aspect transformations that result from primitive tracecuts a and b

```
private final class Urd$a$Token extends Token {
    private String s;
    public  Urd$a$Token(String a0){
        s = a0;
    }
    public static int GetSymbolId(){
        return 1;
    }
    public int get$s{
        return s;
    }

};

private final class Urd$b$Token extends Token {
    public static int GetSymbolId(){
        return 2;
    }
};
```

Figure 21: Token class generated for each primitive tracecut

Instrumentation is achieved through basic constructs present in the AspectJ language. Recall that AspectJ provides a means to augment existing code to provide different or additional behaviour. Before advice allows code to be added before a join point of interest executes; similarly, after advice provides for additional code after a join point of interest has executed.

```
tracecut a() {: String temp :} ::=  entry(safePc(..) && args(temp)) {:
   if(temp.length() == 0 || temp == null){
     fail;
   }
:}
```

Figure 22: A modified version of primitive tracecut `a` so to demonstrate the transformations and generation of code the result when semantic actions are associated.

When a primitive `tracecut` employs some semantic constraint on the acceptance of an event, that semantic constraint is added to the generated advice block. If **`fail`** is used, all occurrences of **`fail`** are transformed into **`return`**–effectively cutting short the advice before the token is feed to the parser. Figures 22 and 23 show the translation of a primitive tracecut with a semantic constraint. In Figure 22, we modify `a` so that it checks its argument to see if it is the empty string. If the argument is empty, the event is not valid. Note in Figure 23 that the `fail` keyword, when used in primitive tracecut declarations is directly translated into `return`, forcing the advice to cut short its execution, avoiding the emission of a token to the parser. Also notice that the temporary variable `temp` is integrated into the signature of the before advice and bound to the exposed argument just as a normal formal parameter would have been.

```
before(String temp): safePC(..) && args(temp) {
   if(temp.length() == 0 || temp == null) {
       return;
   }
   getParser$0().next(new Urd$a$Token());
}
```

Figure 23: Generated code when semantic constraints are applied to primitive tracecut `a`.

4.2.5 Ordered Tracecuts

Ordered tracecuts form the grammar rules of a declarative event pattern. During the translation process this variant of tracecut essentially disappear; for each history target tracecut, a transitive closure is performed to determine all relevant tracecuts for a given target. From those remaining tracecuts, an abstract representation of the grammar is constructed. This grammar is then used as the basis for constructing event parsers. The details of this process can be found in Section 4.3.

However, when an ordered tracecut exposes state through formal parameters, an intermediate node must be generated to hold the context of interest. This process is analogous to the generation of Token classes for each primitive tracecut. Figures 24 and 25 demonstrate this generation process for a modified `isSafe` tracecut. We assume a modified `a` primitive tracecut which exposes a string value.

These intermediate nodes are instantiated as needed by the parser to maintain ex-

```
tracecut isSafe(String s)::= a(s) completed()* $;
```

Figure 24: Ordered tracecut `isSafe` before translation

```
private final class Urd$isSafe$Intermediate extends Node {
   private String s;
   public Urd$isSafe$Intermediate(String a0){
        s = a0;
   }
   public String get$s(){
        return s;
   }
};
```

Figure 25: Ordered tracecut `isSafe` after generation process

posed state for later use. Similarly, the semantic code associated with an ordered tracecut is executed at specific points in the recognition of the pattern. See Section 4.4.3 for more details.

4.3 Event Parsers

In this section we discuss our approach for parsing languages described by context-free grammars, the formalism which our tracecut specifications are based on. In particular we look at a variant approach to LR parsing that we refer to as *reduced stack-activity parsing*. In Section 4.3.1 we begin by discussing reduced LR parsers in general. Using the DEP example in Figure 15 to motivate the benefits of stack-reduced parsers, we first demonstrate how a conventional LR parser technique (LALR(1)[1]) recognizes a stream of events. In Section 4.3.2 we outline the construction of stack-reduced parsers. We then constrast the LALR(1) approach with how equivalent stack-reduced parsers perform the same recognition task in Section 4.3.10.

4.3.1 LR Parsing

LR parsers [7] use the stack extensively to memorize the left context of the parse; in fact in the worst case, the stack may contain the entire input before any reduction can occur. While this may be acceptable in cases where the input stream is fixed, for example, when parsing the fixed contents of a source code file, it is undesirable for parsing the potentially infinite stream of events emitted from a program execution. To illustrate this, we will consider a traditional LALR(1)[5] parser for our example.

Figure 26 shows a grammar derived from our example in Section 4.1.1. The tracecut names present in the example in Figure 15 have been abbreviated for the sake of clarity (*Is* == `isSafe`, *Cs* == `completed*`, *C* == `completed`, *a* == `entry(safe())`,

[5]LALR(1) parsers are those generated by common parser generator tools[1], and used to parse the vast majority of modern programming languages

b == `exit(safe())`, *c* == `entry(unsafe())`, and *d* == `exit(unsafe())`). In this figure, each rule corresponds to a tracecut clause. We use upper-case letters to denote non-terminal symbols (Ordered tracecuts) and lower-case letters to indicate terminal symbols (primitive tracecuts).

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

Figure 26: Grammar for our running example

The LALR(1) parser automaton for this grammar is is presented in Figure 27. The construction of this parser automaton follows conventional approaches found in most textbook references on compiler construction [1].

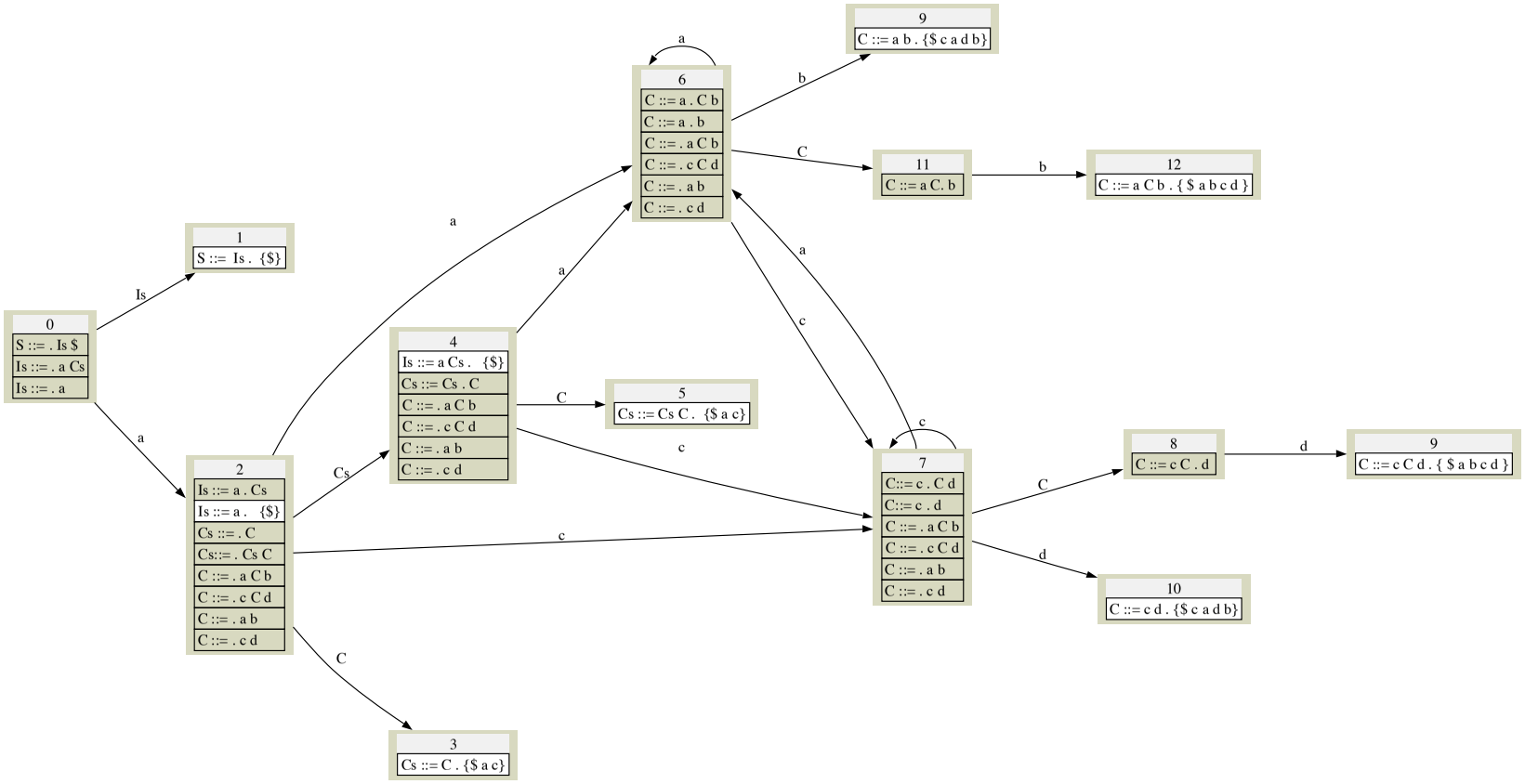

Figure 27: LALR(1) DFA for grammar. Reductions are highlighted. Lookahead sets are enclosed in braces () for reductions

Intuitively, an LALR automaton (such as the one in Figure 27) represents the deterministic choices a parser will make while recognizing viable prefixes. This choice is based on the current state (state on top of the stack) and knowledge of the next symbol of input. In conjunction with a single stack, this automaton is used to properly recognize or reject a stream of input. The stack is used to accumulate grammar symbols and the states corresponding to these symbols as a parse progresses. Starting at some start state, the next symbol of input is pushed or *shifted* onto the stack if, from the current state, a transition is labelled with that symbol.

If there is no transition leading from the current state and having the label of the next symbol, the input may not be valid, but it might be the case that a grammar rule *reduction* is possible. A reduction will occur if the next symbol of input is a member of the look-ahead set of a rule in the grammar. In Figure 27, the valid reductions from each state are highlighted and the look-ahead sets are shown in parentheses following the rule. For example, in state 9 of Figure 27, if the next symbol of input is in the set {$ a b c d}, then rule 7 of the grammar (Figure 26) will be reduced. When a reduction occurs, those symbols on the top of the stack corresponding to the grammar rule being reduced are popped, the current state becomes that of the new top of the stack, and the next symbol is that of the non-terminal grammar symbol of the rule being reduced. if a transition labelled with the new (non-terminal) next symbol leads from the current state (the top of the stack), the new next symbol is pushed onto the stack and the transition is followed, just as before.

When presented with the a typical steam of events ($a \longrightarrow a \longrightarrow c \longrightarrow d \longrightarrow b \longrightarrow \$$), the LALR(1) parser for out grammar yields a traversal as shown in Table 1.

Stack	Remaining Input	Action
0_ϵ	a a c d b $	
$0_\epsilon\ 2_a$	a c d b $	s2
$0_\epsilon\ 2_a\ 6_a$	c d b $	s6
$0_\epsilon\ 2_a\ 6_a\ 7_c$	d b $	s7
$0_\epsilon\ 2_a\ 6_a\ 7_c\ 10_d$	b $	s10
$0_\epsilon\ 2_a\ 6_a\ 11_C$	b $	r8 ($C \rightarrow c\ d$)
$0_\epsilon\ 2_a\ 6_a\ 11_C\ 12_b$	$	s12
$0_\epsilon\ 2_a\ 3_C$	$	r5 ($C \rightarrow a\ C\ b$)
$0_\epsilon\ 2_a\ 4_{Cs}$	$	r4 ($Cs \rightarrow C$)
$0_\epsilon\ 1_{Is}$	$	r1 ($Is \rightarrow a\ Cs$)
$0_\epsilon\ 1_{Is}$ $		s-$
$0_{S'}$		r0 ($S' \rightarrow Is$ $) ACCEPT

Table 1: Traversal of LALR parser for grammar on input ($a \longrightarrow a \longrightarrow c \longrightarrow d \longrightarrow b \longrightarrow \$$)

Table 1 shows the parsing steps involved for the recognition of a short sequence of inputs ($a \longrightarrow a \longrightarrow c \longrightarrow d \longrightarrow b \longrightarrow \$$). If one was to follow this traversal, they would first see that on an input of the symbol a in state 0, the input (and state) would be shifted onto the stack, making the current state 2. Similarly, the next a causes another shift onto the stack and brings the current state to 6. On input c a shift occurs. Symbol d is then shifted onto the stack, making the current state 10. We are now able to reduce when presented with input b, as b is a member of the lookahead set of rule 8 from state 10. The top of the stack is popped for each of the symbols in rule 8 (both 7_c and 10_d), leaving the current state 6. From state 6, a transition exists over symbol C, which is

the next symbol as a result of the reduction. The parse proceeds in this manner until finally reaching the end of input, where then a series of reductions are made before finally arriving again at state 0 and having reduced the rule 0.

One should note that the stack column in this table shows the frequent use of the stack for the accumulation and reduction of input state. The excessive use of the stack lends itself to slower parsing performance, as compared to say, deterministic finite automata, which do not rely on a stack, but which cannot recognize the range of languages that LR parsing can. In the next section we discuss reduce stack-activity parsers, and show how they can avoid the use of the stack whenever possible, resulting in faster parsers.

4.3.2 Reduced Stack-activity Approach

Aycock/Horspool [3, 2] and later Scott/Johnstone [11, 9, 8, 10] have proposed what we call here reduced stack-activity parsers; these parsers aim to limit the use of the stack by encoding as much left context information as possible in the automaton itself. These approaches isolate the recursive nature of the grammar, and resort to the stack only when required. Both differ from more traditional LR parsing approaches in that the resulting parsers consist of a hierarchy of regular automata; when an inherently recursive part of the grammar is encountered, calls are made into a sub-automaton solely dedicated to handling the recognition of the recursive fragment. When the recursive fragment is recognized, control returns to the calling context. The approach mirrors subroutines in a programming language, and employs a call stack in the same fashion. In the following sections we provide an overview, and then demonstrate how, unlike the LALR example above, these approaches avoid the use of the stack whenever possible, resulting in improved parse performance.

4.3.3 Derived grammar

In the Aycock/Horspool approach, parser construction begins with an analysis of the given grammar to find the points where recursion occurs. From this analysis, an augmented grammar representing all the possible left contexts of each non-terminal symbol is created. This augmented grammar is then used in the generation of a novel parsing automaton employing a trie based construction. In this section we outline the procedure using our grammar (see Figure 26).

The grammar analysis aims to derive a new grammar F which will specify the left context of each of the non-terminals from the original grammar G (Figure 26). The set of terminals in F include both the set of terminals and non-terminals of G. F's non-terminals are derived (but not the same as those) from the non-terminals of G. These derived non-terminals are enclosed in square brackets to mark their significance (e.g.. non-terminal $[A]$ derived from A).

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

of $[A]$ is whatever can come before $[B]$ followed by α. Finally, F is made tidy by removing direct cycles such as $[A] \rightarrow [A]$.

The derived grammar for our running example is show in figure 28. By construc-

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

Figure 28: Derived Grammar for our running example

tion the grammar F describes a regular (left-linear) language. Because F is a regular grammar, it is equivalent to a FA. The FA for F of our running example is shown in Figure 29. Finding recursive points in the original grammar results from considering that a FA will contain a non-ϵ cycle if and only if the FA accepts an infinite language. By selecting a minimal set of transitions from the FA so that such cycles are then broken we can locate the points in the left context of grammar were right recursion is exhibited. Aycock/Horspool point out that such a selection is equivalent to the Feedback arc set (FAS) problem and that it is an NP-complete problem. While they point to heuristic algorithms to FAS, since *URD* is a proof-of-concept we simply remove all back-edges from the graph to break cycles. The FA for the F of the running example is show after removal of the feedback arcs in Figure 30.

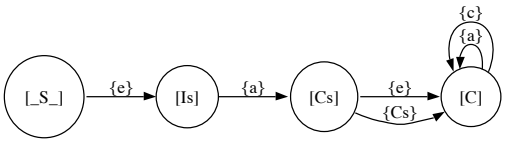

Figure 29: Left Context analysis

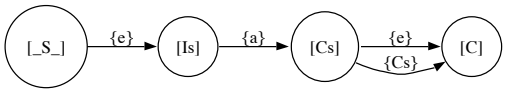

Figure 30: Augmented Left Context analysis. Recursion is removed.

4.3.4 Limit points and Augmented grammar

The broken cycles in the augmented grammar F (see Figures 29 and 30) are used to determine points in the original grammar G that lead to unavoidable recursion. Using

Figure 30 as a guide, we can alter the original grammar to indicate these recursive points. These *limit points* in the grammar are so named because they indicate the points in the grammar where a finite memory reaches its limit and from where on a pushdown must be used.

Let $\phi_{source} \xrightarrow{\alpha} \phi_{target}$ be an transition between states *source* and *target* where α is some string of symbols labelling this transition. For each removed edge from the DFA in Figure 29, and for each production rule in G for non-terminal equal to ϕ_{source}, mark occurrences of the label of ϕ_{target} on the right-hand side or the rule with the a special symbol lp, but only if preceded by the string of symbols α. In the case of our running example, the grammar in Figure 31 results.

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

Figure 31: Augmented Grammar for our running example

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

4.3.5 Viable Prefix (Φ) set

Having determined the recursive nature of the original grammar, and having marked such points in the grammar, we can now combine the limit-point grammar together with the augmented grammar (Figure 30) to generate the viable prefixes of each of the non-terminal symbols in the original grammar. Because cycles have been removed from Figure 30, the resulting generation of left-contexts is finitely large. The set of left-contexts combined with the *handles* (right-hand side of a grammar rule) of each non-terminal symbol (referred to here as the Φ set), is generated. Table 2 shows the left contexts and viable prefixes for the non-terminals in our running example.

4.3.6 Parsing automata construction

A trie is a tree data structure for storing a set of strings in which there is one node for every shared prefix. The construction uses this structure as the basis for the parsing automata. Those strings in the Φ set are added to form a trie. Figure 32 shows the end result. With some effort one can see that the viable prefixes listed in Table 2 are fully represented in the trie structure.

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

Table 2: Φ Set for our running example

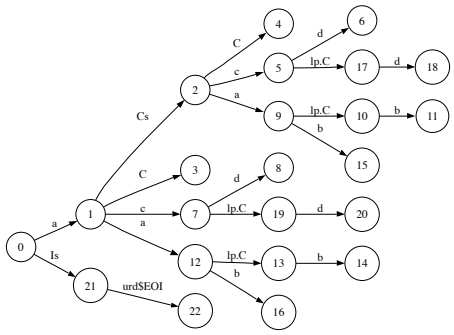

Figure 32: Trie

Special reduction edges are then added to the trie to recognize the occurrence of a given rule in the grammar. For each viable prefix, a reduction edge is added from the ending state of the prefix back to a state reachable by a transition on the non-terminal of the prefix, but only if such a state originates from a state reachable by tracing a handle of the non-terminal backwards from the end of prefix state. The reduction edge is then labelled to indicate the handles corresponding production rule. For example, a reduction edge is added from state 20 to state 3, and is labelled REDUCE 5, because a backwards path from state 20 (the end state of a viable prefix of C) matches the handle of rule 5 ($C \rightarrow a\ lp.C\ b$). Figure 33 depicts the trie with reduction edges added.

4.3.7 Sub-automata

The limit points in the grammar are those points where the use of the stack is unavoidable. Analogous to subroutines in a programming language, each unique limit point is expanding into a separate sub-automaton that matches a subset of the language. A

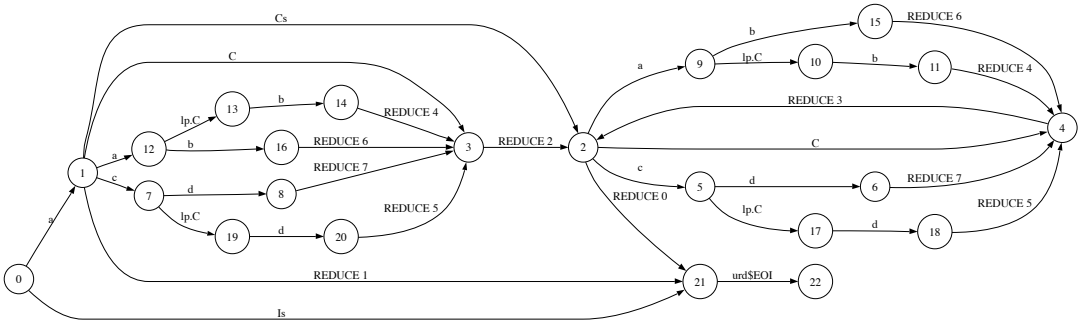

Figure 33: Reduction Edges

new grammar is derived from the original grammar for each non-terminal represented by a limit point. This grammar contains those parts of the original grammar reachable from its corresponding non-terminal. From these new grammars, sub-automata are constructed through the same process described above. Figure 34 shows the automata for limit point C, as well as for our original target non-terminal Is. Transitions over $lp.C$ are dotted in order to highlight these recursive points.

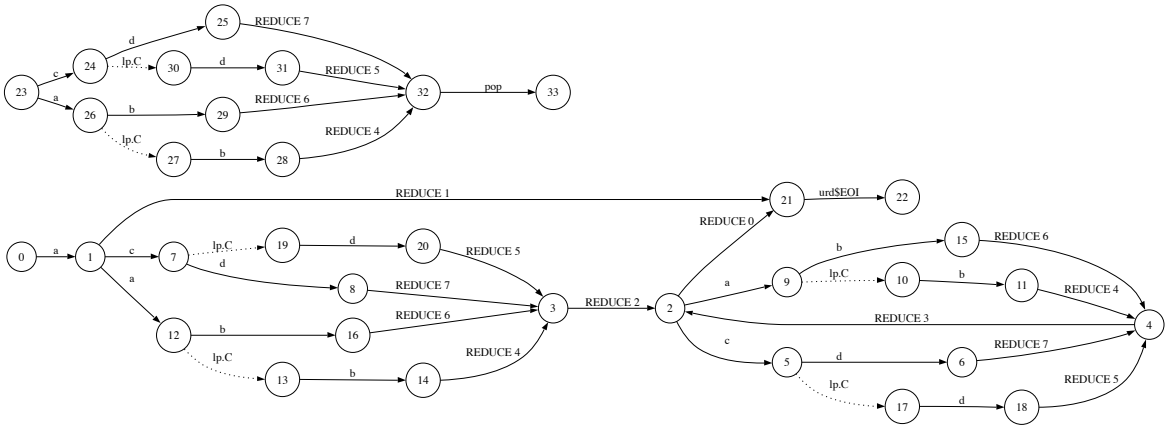

Figure 34: Automaton for limit point $lp.C$, as well as for our original target non-terminal Is. Transitions over $lp.C$ are dotted in order to highlight these recursive points.

4.3.8 Lookahead calculation

The LR(0) state machines that results from the construction process now await the addition of lookahead information. Our approach attempts to disambiguate when possible with a single lookahead. To this end, we attempt to calculate the LALR lookahead

sets for the LR(0) machine. When this lookahead information fails to disambiguate, we simply fall back on the non-deterministic natural of this general parsing algorithm. The lookahead information aims to lessen the ambiguity, and so to the work that must be done. We employ the approach to the computation of LALR lookahead sets as described in [4].

4.3.9 Nonterminal Edge Removal & Limitpoint Correction

Once reduction edges are added, and lookahead calculated, the non-terminal edges are removed, leaving a complete parsing automaton. Remaining limit point edges are converted to *calling* edges—resulting in a pushing of current state to the stack and resumption of regular recognition is a sub-automaton. *Returning* pop edges are added to return back to the calling context when the sub-automaton has completed. The final automaton is shown in Figure 35.

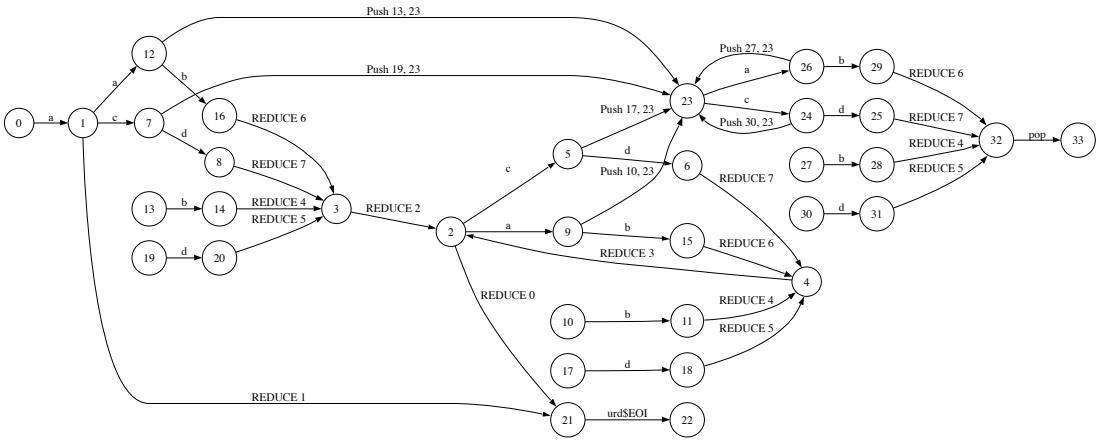

Figure 35: Final parsing automaton. Limit point transitions have been expanded

4.3.10 Comparison with LALR parser

Comparing the traversals of reduced-stack-activity parser with the LALR parser shown in Table 1, we can see an immediate improvement in the amount of stack use, even for the short stream of events we have used.

Table 4 shows the traversal through the automaton shown in Figure 35, when presented with the following steam of events.

$$a \longrightarrow a \longrightarrow c \longrightarrow d \longrightarrow b \longrightarrow \$$$

This is the same stream of events the LALR(1) parser recognized earlier. We have repeated the LALR(1) traversal in Table 3 for easy comparison.

Even for a small stream of events, such as the one used above, the benefits of stack-activity reduced parser approach are evident. Where in Table 3, a stack operator

Stack	Remaining Input	Action
0_ϵ	a a c d b $	
$0_\epsilon\ 2_a$	a c d b $	s2
$0_\epsilon\ 2_a\ 6_a$	c d b $	s6
$0_\epsilon\ 2_a\ 6_a\ 7_c$	d b $	s7
$0_\epsilon\ 2_a\ 6_a\ 7_c\ 10_d$	b $	s10
$0_\epsilon\ 2_a\ 6_a\ 11_C$	b $	r8 ($C \rightarrow c\ d$)
$0_\epsilon\ 2_a\ 6_a\ 11_C\ 12_b$	$	s12
$0_\epsilon\ 2_a\ 3_C$	$	r5 ($C \rightarrow a\ C\ b$)
$0_\epsilon\ 2_a\ 4_{Cs}$	$	r4 ($Cs \rightarrow C$)
$0_\epsilon\ 1_{Is}$	$	r1 ($Is \rightarrow a\ Cs$)
$0_\epsilon\ 1_{Is}$ $		s-$
$0_{S'}$		r0 ($S' \rightarrow Is$ $) ACCEPT

Table 3: Traversal of LALR parser for grammar on input ($a \longrightarrow a \longrightarrow c \longrightarrow d \longrightarrow b \longrightarrow \$$)

Stack	State	Remaining Input	Action
	0_ϵ	a a c d b $	
	1_a	a c d b $	s1
	12_a	c d b $	s12
13	23	c d b $	push 13, goto23
13	24_c	d b $	s24
13	25_d	b $	s25
13	32	b $	r7
	13	b $	pop
	14_b	$	s14
	3	$	r4
	2	$	r2
	21	$	r0
	$22_\$$		accept

Table 4: Traversal of reduced-stack activity parser for grammar on input ($a \longrightarrow a \longrightarrow c \longrightarrow d \longrightarrow b \longrightarrow \$$)

occurred at essentially each step of the traversal, the reduced-stack activity parser only requires the use of the stack once. This thifty use of the stack amounts to improved runtime performance; an important goal for our tool.

4.4 More on Traversals

Although the above grammar is deterministic, in general the traversal of the generated parsing automaton may be non-deterministic. A *general* parsing approach is necessary since we have not imposed any limitations on the grammars underlying tracecuts—tracecuts may describe *any* context-free grammar (not only unambiguous ones). In this Section 4.4.1 we give an overview of the general traversal strategy in use. Exposure and forwarding of context is the topic of Section 4.4.3, where we discuss the operation of the *semantic stack*. The presence (or absence of) anchoring tracecuts alter facets of the traversal; we look at these subtleties in Section 4.4.2. Similarly, the `fail` keyword, when encountered within the semantic code block of an ordered tracecut, influences the traversal in ways discussed in section 4.4.4.

4.4.1 Processing elements

In a completely deterministic automaton there is ever only one processing element. It proceeds along the deterministic paths set forth by the automaton, always able to select the correct alternative edge whenever faced with an option. However, in a general parsing traversal (such as the one we use), when presented with a non-deterministic choice of which path to follow, a new processing element is duplicated for each each possible choice and the traversal proceeds in a breadth-first fashion. In this way, all potential interpretations of the grammar are considered. Whenever the parser observes an event, it is processed by each processing element. The more deterministic the grammar, the less processing work there is to conduct at each step.

Associated with each processing element is a parse stack. When a processing element follows a *push* label, the state which the processing element should resume at is pushed to the stack. Pop transitions cause this stack to be popped, returning the processing element to the popped state before continuing. When duplication of processing elements occurs, strictly speaking, duplication of this parse stack is required. However, a *graph-structured stack* can be used to avoid much of this duplication [12].

It is often the case that a divergent processing element may quickly exhaust an exploration path and arrive at a dead end. This is the case whenever a terminal symbol is observed but not expected, and no reduction is possible. In general, a processing element is said to die whenever there are no longer any viable directions for it to follow. The death of a processing element involves the removal of the element from active processing, and the deconstruction of its unshared portion of parsers pushdown.

It may be the case that all processing elements of an automaton have died. Depending on the anchoring symbols employed, this may yield either a target tracecut that never will match, or conversely, one that has been satisfied and that will always match, now and further along in the execution of the program.

4.4.2 Anchor effects on traversals

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

When a pattern is anchored to the beginning of the trace, there is no opportunity for the the traversal to *reset*—no new generation of processing elements will ever be created to recognize the pattern anew.

The absence of the beginning-of-trace anchor permits the pattern to be matched beginning at any point in the trace—this amounts to the introduction of a new processing element whenever any symbol that could begin the target tracecut is observed.

The occurrence of the end-of-trace anchor in a pattern indicates that, whenever queried as to whether the pattern is current satisfied, a processing element should be able to reach the end state of the automaton and provide a disambiguated reading. When presented with the end-of-trace symbol, processing elements are introduced to follow all transitions from current states to those that contain the end-of-trace symbol in their follow sets. More than one of these processing elements may arrive at and

converge on the final state; the last to arrive is made the currently satisfying processing element.

When the end-of-trace anchor is absent from a pattern, this indicates that the pattern may have matched in the past; the parser will maintain the most recent of match. In order to obtain this behaviour, reduction transitions with the end-of-trace symbol in their follow sets must always be traversed after normal processing of a symbol occurs whether the pattern is queried for acceptance or not. This additional expressiveness incurs the extra processing costs associated with it.

In the absence of a caret or dollar sign in a DEP, the DEP will be matched greedily from the beginning of the communication history. New processing elements will be generated on the occurrence of first events, and end-of-trace will be challenged after each normal processing round.

4.4.3 Semantic Stack

Whenever a processing element encounters an event which exposes state for use in the tracecut, it stores this state (wrapped within a token instance) on a stack known as the *semantic stack*. When a processing element follows a reduction, its semantic stack may be reduced and exposed state (wrapped in token or intermediate node class instances) may be manipulated by code in the semantic blocks of ordered tracecuts. Processing elements share their semantic stacks in a manner parallel to the way they share their parsing stacks. When these reductions occur, additional code is executed for each ordered tracecut and associated semantic action. Figure 36 shows an ordered tracecut over two named tracecuts `a` and `b`. The tracecut `a` exposes an integer which is bound to the formal `i` in this example. Figure 37 shows how this is translated into action code executed in step with reductions.

```
tracecut example(Integer i) ::= a(i) b() {:
    if(i.intValue() < 0){
            fail;
    }
:};
```

Figure 36: An example of an ordered tracecut using fail.

A case is defined for each rule in the tracecut grammar. Exposed state is referenced from the Semantic stack `stack`, and semantic action code is then executed. Finally, a new intermediate node is instantiated to carried the exposed state forward, and and this construction is returned. Assuming the call to the reduceAction method returns a non-null value, the semantic stack is adjusted; and parsing commences. If the method returns null, however, this indicates a failure.

4.4.4 More on fail

The behaviour of the keyword `fail`, when it appears in the semantic action block of an ordered tracecut, should now be clear. When `fail` occurs in the semantic action of an ordered tracecut, it is directly transformed into `return null;` (see Figure 37).

```
public urd.deps.runtime.Urd$Node reduceAction$example(SemStack stack,
       int urd$rule) {
       switch (urd$rule) {
           case 0:
               Urd$a$Symbol pos0 = (Urd$a$Symbol) stack.peek(0);
               Integer i = pos0.i;
               if (i.intValue() < 0) { return null; }
               Urd$example$Interm urd$result =
                 new Urd$example$Interm(i, 1, urd$rule);
               return urd$result;
           default:
               return null;
       }
```

Figure 37: The generated reduction code for the example tracecut in figure 36.

If reduceAction returns a null, the current processing element is set to die, the path it was exploring is made invalid without any recourse for recovery.

5 Discussion

Many factors contributed to our choice to use AspectJ as a foundation for experimentation with DEPs. AspectJ is arguably the most well-known, and well-supported AOP realization available; a large base of users from both research and industrial are familiar with its syntax and capabilities. As a language, AspectJ already provides sufficiently low-level building blocks (namely, pointcuts and advice) on which to support the higher-level DEPs abstractions.

Rather than provide declarative event patterns atop AspectJ, we could have chosen a different base or to start from scratch. The combination of AspectJ and DEPs is imperfect. DEPs operate on events while AspectJ operates on join points. What constitutes a join point depends on the join point model in sway for a given language. In AspectJ, method execution is considered a join point, but it is not an event in the sense described above since it cannot be instantaneous. It is the combination of advice and pointcut that determines when an “event” really happens. This difference can lead to subtle problems; however, this does not imply a shortcoming in the declarative event patterns approach. An industrial-strength realization of DEPs would need to reconcile the differences more cleanly.

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.

6 Related Work

Various techniques make use of event traces or historical references for purposes other than implementation. In the conference paper for which this report is companion to, these works are discussed and compared. The interested reader is referred to [15].

7 Conclusion

This paper has presented an initial implementation of declarative event patterns, a language that allows for patterns in a program's execution to be expressed as context-free patterns of events, and for the occurrences of these patterns to alter the course of the program's execution. We have explored an initial realization of our declarative event pattern language that leverages the power and applicability of aspect-oriented programming (AOP). We have added to AspectJ two straight forward language constructs to support the recognition of patterns of events. Our proof-of-concept tool transforms AspectJ code augmented with our new constructs into programs implemented in standard AspectJ, able to recognize and react to patterns of events when they occur in the execution of a program. The details of these extensions and how our tool concretely realizes them have been shown. Our tool takes advantage of recent advances in the recognition of context-free languages, using a general parsing strategy to accommodate unconstrained context-free patterns; this expressiveness is not without a cost, but importantly, these costs are only incurred when this added expressiveness is used. The *URD* tool presented in this report is a proof-of-concept; many improvements are still possible.

References

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

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

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

[4] M. E. Bermudez and G. Logothetis. Simple Computation of LALR(1) Lookahead Sets. *Information Processing Letters*, 31:233–238, June 1989.

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

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented*

Programming, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997.

[7] D. E. Knuth. On the translation of languages from left to right. *Inform. Control*, 8:607–639, 1965.

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

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

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

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

[12] M. Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, 1986.

[13] R. J. Walker. *Essential Software Structure through Implicit Context*. PhD thesis, Department of Computer Science, University of British Columbia, February 2003.

[14] R. J. Walker and G. C. Murphy. Implicit context: Easing software evolution and reuse. In *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*, pages 69–78, 2000.

[15] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT Twelth International Symposium on the Foundations of Software Engineering*, Newport Beach, CA, USA, October/November 2004. ACM Press.