Test-Based Pointcuts for Robust and Fine-Grained Join Point Specification

Kouhei Sakurai
Graduate School of Arts and Sciences,
University of Tokyo
sakurai@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
Graduate School of Arts and Sciences,
University of Tokyo
masuhara@acm.org

Abstract

We propose *test-based pointcuts*, a novel pointcut mechanism for AspectJ-like aspect-oriented programming languages. The idea behind the test-based pointcuts is to specify join points through unit test cases associated with the target program. The test-based pointcuts improve robustness and precision of pointcut languages. The test-based pointcuts are more robust against software evolution because they do not directly rely on identifier names in a target program. The test-based pointcuts are more precise because they can distinguish fine grained execution histories including conditional branches by comparing the runtime execution histories with recorded for ones of the unit test cases. This paper presents design and implementation of the test-based pointcuts as an extension of an AspectJ compiler. We evaluated robustness and runtime efficiency of test-based pointcuts through case studies that applied test-based pointcuts to several versions of practical application programs.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords Test-based Pointcuts, Fragile Pointcut Problem, Aspect-oriented programming language, Unit Test Cases

1. Introduction

Aspect-oriented programming (AOP) [18, 19] is a programming technique in order to modularize crosscutting concerns. In AOP, programmers can define an aspect as a modular unit of the crosscutting concern with respect to a target program[1]. Pointcut languages are a key abstraction mechanism for aspects that allows the programmer to specify which join points (i.e., runtime actions of the target program) match. There have been many studies on pointcut languages in order to improve their precision [2, 9, 10, 11, 15, 22, 23, 27], robustness [1, 3, 8, 12, 13, 17, 21, 22], understandability [14], and safety [4].

[1] In this paper, a target program means a program affected by the aspects.

This paper attempts to improve pointcut languages in terms of two properties, namely *robustness* and *precision*, which are difficult to be satisfied at the same time in existing pointcut languages.

Robustness of a pointcut is a property whether the set of join points matched by the pointcut meets the developer's intention after software evolution. It is known that the property is hard to be guaranteed in existing pointcut languages, as also known as the fragile pointcut problem [17, 26]. This is because in existing pointcut languages, pointcuts tend to directly depend on implementation details of a target program, such as the signature of a method called inside of another method. Existing proposals to this problem include the ones to limit scope of pointcuts (e.g., Crosscutting Programming Interface (XPI) [12] and Open Modules [1, 21]), the ones to exploit information at not only an implementation phase of the software development, but also other phases such as a design phase (e.g., model-based pointcuts [17] and Motorola WEAVER [8]), or the ones to exploit program information at higher level (e.g., analysis-based pointcuts [3, 13, 22]).

Pointcut languages should be *precise* enough so that programmers can exactly specify the set of join points that are affected by an aspect. In this regards, pointcut languages have been improved so that they can distinguish calling context (e.g., `cflow`), execution history (e.g., tracematchs [2] and tracecuts [27]), intra-procedural control flow (e.g., loop and conditional branch join points [15, 23]), and so forth.

In existing AOP languages, it is often difficult to satisfy those two desired properties. More precise pointcuts tend to be more dependent on the implementation details, which is less robust. For example, assume the programmer wants to apply advice when a method behaved in a specific way. With existing techniques, he or she has to specify join points performed inside the method to be advised. The resulted pointcut is less robust because it depends on the internal implementation of the method, which tends to change more frequently than public interface of a module.

In order to resolve this dilemma, this paper proposes a novel pointcut mechanism called the *test-based pointcuts* with which the programmer can specify more precise pointcuts in more robust manner. Concretely, a test-based pointcut can distinguish different histories without directly depending on detailed implementation of a method. The underlying idea is to specify execution history through examples of program executions. We use unit test cases as example programs, which can be easily maintained along with software evolution. Our current design treats a set of runtime actions during a method execution as an execution history. A history contains information of branches taken at conditional branches, which thus make the pointcuts more precise than the other existing history sensitive pointcuts.

The rest of the paper explains our proposal in the following order. Section 2 gives a motivating example where we need precision

```
class FtpConnection {
   boolean chdir(String dir) {...}
   String[] list() {...}
   void upload(String file) {...}
   void download(String file) {...}
   ...//other methods such as login follow here
}
```

Figure 1. Outline of the `FtpConnection` class

and robustness at the same time. Section 3 reviews a unit testing framework that we use for defining pointcuts. Section 4 introduces the test-based pointcuts and describes its semantics. Section 5 describes the how current implementation efficiently match pointcuts. Section 6 evaluates how test-based pointcuts can replace and improve existing pointcuts through case studies. Section 7 discusses related work. Section 8 concludes the paper.

2. Motivating Example

This section explains precision and robustness properties of pointcuts by using concrete code. We take, as an example, a GUI updating concern in an AOP version of JFtp[2], which is a network file browser. The program is obtained by simplifying and refactoring the original JFtp program in Java into an AspectJ program while preserving its behavior.

2.1 Structure of AO-Refactored JFtp

The core part of JFtp we focus on here consists of the `FtpConnection` class, the `DirPanel` GUI component and the `ConnectionUpdate` aspect. `FtpConnection` represents network connections with an FTP server. The GUI component displays files in the current remote directory. `ConnectionUpdate` notifies the GUI component of changes of the server and connection states.

2.1.1 The `FtpConnection` Class

`FtpConnection` objects (Figure 1) represent connections to an FTP server. The methods defined in the `FtpConnection` class implement all operations of the file transfer protocol. For example, the `chdir` method takes a remote directory name and sends a command to the server in order to change the remote working directory.

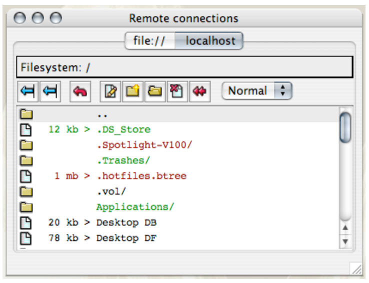
2.1.2 GUI Component

Among many GUI components in JFtp, the `DirPanel` GUI component displays the list of files in the current working directory on the connected server as shown in Figure 2. Figure 3 shows an outline of the definition. The method `updateRemoteDirectory` updates the list of files by calling `setDirList` and `revalidate`.

2.1.3 Notification Aspect in Current AspectJ

The `ConnectionUpdate` aspect shown in Figure 4 implements the GUI updating concern that refreshes the GUI components whenever the server status or connection states change. It crosscuts most operations in `FtpConnection`. For example,the `DirPanel` component must be updated when the current working directory changes. The aspect therefore defines the `updateRemoteDirectory` pointcut that matches the method execution join points of `chdir` method in the `FtpConnection` class (lines 2 to 4 in Figure 4), and an `after` advice declaration that updates the file list by calling the `updateRemoteDirectory` method of the `DirPanel` component that is associated with the connection obtained from a `target` pointcut (line 5 to 10 in Figure 4).

[2] `http://j-ftp.sourceforge.net/`



```
Filesystem: /
          ..
12 kb > .DS_Store
        .Spotlight-V100/
        .Trashes/
 1 mb > .hotfiles.btree
        .vol/
        Applications/
20 kb > Desktop DB
78 kb > Desktop DF
```

Figure 2. Screenshot of a GUI component in JFtp

```
class DirPanel extends JPanel {
   ...
   public void updateRemoteDirectory() {
     setDirList(); //refreshes the list of files
     revalidate(); //cause updating
                   // graphics of the component
   }
   ...//the definition of setDirList() follows here
}
```

Figure 3. Outline of a GUI component in JFtp

```
1  aspect ConnectionUpdate {
2     pointcut updateRemoteDirectory():
3       execution(boolean
4             FtpConnection.chdir(String));
5     after(FtpConnection c):
6      updateRemoteDirectory() && target(c) {
7       //refreshes layout of a GUI component
8       // associated with the connection
9       c.panel.updateRemoteDirectory();
10    }
11    ...//code for managing association between
12       // a connection and a GUI component follows here
13 }
```

Figure 4. Outline of the `ConnectionUpdate` aspect

2.2 Robustness Against Software Evolution

The pointcut definition in `ConnectionUpdate` is not robust against software evolution as it directly relies on the identifier names in the target program. The problem is also known as the fragile pointcut problem: when a developer changes a target program without knowing the pointcuts in the aspects, the change may make join points accidentally matching or unmatching and result in an unintended program behavior.

As a concrete example, we investigate the evolution of the `FtpConnection` class. From version 1.07 to version 1.15, as shown in Figure 5, the class is evolved to an interface `BasicConnection` and a sibling class `FilesystemConnection` so that the new version can also handle local files in the same manner as the files on FTP servers.

The pointcut in the `ConnectionUpdate` aspect is not robust against this evolution. The intention of the GUI updating concern is to update the file listing component when the current directory changes. Therefore, the `updateRemoteDirectory` pointcut should also match execution of `chdir` of the `FilesystemConnection` class. However, it does not because this change was not anticipated when the pointcut was written.

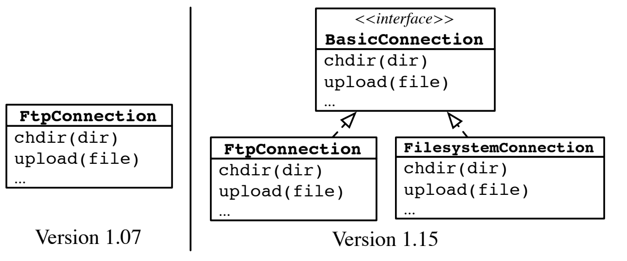


Figure 5. Evolution of the `FtpConnection` class

```
 1  public boolean chdir(String n) {
 2    try {
 3      socketio.send("CWD " + n);
 4    } catch (Exception e) {}
 5    String l = socketio.readLine();
 6    if (l == null) {
 7      return false; //update
 8    } else if (l.startsWith(SUCCESS)) {
 9      return true;  //update
10    } else {
11      return false; //no update
12  } }
```

Figure 6. Implementation of the `chdir` method

2.3 Precision in terms of Execution Histories

Another problem of existing pointcut languages is that pointcuts are not sufficiently precise because they can not distinguish fine grained execution histories of join points. To illustrate this point, we continue to use the `ConnectionUpdate` aspect shown in Section 2.1.3.

Assume we would like to avoid updating the file listing when `chdir` failed due to a non existing directory name (resulting in the current working directory to stay the same). In other words, we would like to update only when `chdir` succeeded, or when the server is disconnected during `chdir`. Since the pointcut shown in Figure 4 matches any `chdir` execution, we need to elaborate the definition so that it can distinguish how `chdir` is processed.

This can be achieved if the aspect would observe which internal execution histories that the `chdir` method followed. The `chdir` method implementation is shown in Figure 6 where the `socketio` field stores an object that provides low-level bidirectional connection to the server. An execution of the method can have one of the following three execution histories:

1. *Success*. It sends a command to the server (line 3). Then receives a response from the server (line 5). When it indicates success (line 8), the method returns `true` (line 9).
2. *Failure with a non existing directory name*. After sending the command, it receives a line that indicates failure of the requested operation (line 8). It then returns `false` (line 11).
3. *Failure by a network error*. When the server or the network is down, sending a command throws an exception, which is caught by the handler (line 4). The attempt to receive a line results in a `null` value (line 6). It then returns `false` (line 7).

The current AspectJ's pointcut language is not sufficiently precise. A single pointcut definition cannot distinguish above three cases from a matching join point and its exposing information such as `target` and `args`. For identifying case 1 it needs to check the return values from `chdir` with the help of the `returning` construct and an `if` pointcut, and for identifying case 2 it needs to check the return values from `readLine` that is called from `chdir`.

```
public class TestFtpConnectionChdir
                              extends TestCase {
  FtpConnection con, errorCon;
  protected void setUp() {
    ... //set up host information
    con = new FtpConnection(hostInfoForTest);
    errorCon=new FtpConnection(errorHostInfo);
  }
  protected void tearDown() {
    con.disconnect();
  }
  public void testChdirSuccess() {
    assertTrue(con.chdir("testDir"));
    assertEquals(HOME+"/testDir", con.pwd());
  }
  public void testChdirFail() {
    assertFalse(con.chdir("unknownDir"));
    assertEquals(HOME, con.pwd());
  }
  public void testChdirFailWithNetworkError(){
    assertFalse(errorCon.chdir(""));
    assertEquals("", con.pwd());
  }
}
```

Figure 7. Unit test case methods for the `FtpConnection` class

2.4 Dilemma of Satisfying Robustness and Precision

To summarize the problem, there are two properties that pointcut languages should have. The one is not to directly depend on names of types, methods and fields. The other is the ability to distinguish execution histories including conditional branches in methods. Those requests are hard to be satisfied at the same time with existing pointcut languages.

Solutions with existing techniques are too dependent on the detailed implementation of `chdir`, and thus more fragile against evolution. A solution in AspectJ is to use auxiliary flags and advice declarations. Others are to use extended pointcut mechanisms such as tracematches [2]. With those solutions, the pointcuts and the aspect depend on the fact that `chdir` recognises a network error by the return values from `readLine`, which is merely one possible implementation among others. The aspect would become incorrect if `chdir` was modified to return `false` in the exception handler at line 4 in Figure 6.

3. Overview of Unit Testing

Since our proposal relies on unit testing, we briefly review how unit test cases are written in the JUnit framework[3] .

We refer to a *unit test case* as a test of a behavior of a method in a target program. A behavior of a method is defined as a pair of a set of parameter values and an expected result that is generated from a behavioral specification of the method. A test is a task to validate the behavior of the method by invoking the method with the specified parameters, and confirming if the obtained result meets the expected one. We call the tested method the *target method* of the unit test case.

In the JUnit framework, each unit test case is defined as a separate method in a class.

[3] `http://www.junit.org/`

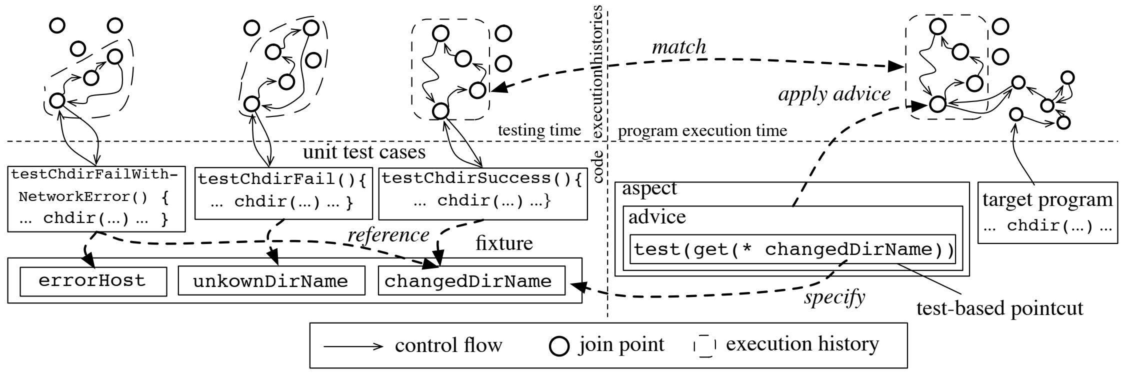


Figure 8. An overview of test-based pointcuts

Figure 7 shows a class containing three unit test cases (`testChdirSuccess`, `testChdirFail` and `testChdirFailWithNetworkError`) along with two auxiliary methods (`setUp` and `tearDown`)[4]. The target method is `chdir` in `FtpConnection`. `testChdirSuccess` (lines 12 to 15) is a unit test case for the case that a change directory operation succeeds. `testChdirFail` (lines 16 to 19) is a unit test for the case that the same operation fails because the directory does not exist on the server. Finally, `testChdirFailWithNetworkError` (lines 20 to 23) is a unit test for the case that the same operation fails due to a network error. The methods whose names begin with `assert`, which are provided by JUnit framework, are used to check whether the results returned from target method are the same to the expected ones.

The methods `setUp` (lines 4 to 8) and `tearDown` (lines 9 to 11) define common tasks at the beginning and the end of each method of a unit test case, which are automatically called by the framework.

4. Test-based Pointcuts

To solve the problems mentioned in Section 2.2 to 2.4, we propose a new mechanism called *test-based pointcuts* as an extension to AspectJ.

4.1 Overview

A test-based pointcut matches join points of the target program through unit test cases. Figure 8 illustrates an overview of the mechanism, which consists of the following three key language elements:

Unit test cases. The programmer defines a unit test case for each execution history that he or she wants to advise. In Figure 8, the three rectangles (`testChdirSuccess`, `testChdirFail` and `testChdirFailWithNetworkError`) in the middle row represent the unit test cases for the `chdir` method. They correspond to the three execution histories discussed in Section 2.3. The unit test cases are defined in the same manner as the ones based on the JUnit framework, and serve as ordinary unit test cases. However, they must call two special methods in order to let the compiler distinguish setup and validation operations from calls of target methods.

Fixture. All unit test cases must access fixtures (i.e., testing values and expected results) through appropriate fields in a class that are defined for and shared by all unit test cases for a target program. This means, for example, the programmer defines the `ConnectionFixture` class for storing fixtures of the three unit test cases in Section 3. Each unit test case that accesses a correct (i.e., existing in the server) directory name uses the `changedDirName` field in `ConnectionFixture`.

[4] This paper assumes JUnit version 3, on which our curerrent implementation depends.

```
pointcut updateRemoteDirectory():
  test(get(* ConnectionFixture.changedDirName)
       ,!within(SocketIO+));
```

Figure 9. A test-based version of `updateRemoteDirectory` pointcut defnition

Test-based pointcuts. A test-based pointcut is written as `test(`p`)` where p is a pointcut description that specifies unit test cases. It matches join points that had the same execution history as the one of the unit test cases specified by p. An optionall syntax `test(`p`,`f`)` lets the `test` pointcut observe only the join points that matched the pointcut f at testing time. This is useful for ignoring the differences between testing and execution environments, such as use of mock objects. This optional syntax will be explained in Section 4.5. For example, `test(get(* changedDirName))` specifies join points that had the same execution histories as `testChdirSuccess` or `testChdirFailWithNetworkError` because they access the field `changedDirName`.

Our system executes a program with test-based pointcuts by following the next two stages:

1. At *testing time* (left hand side of Figure 8), the system compiles the target program without aspects and executes all unit test cases. For each unit test case, the system records an *execution history* of the target method execution (i.e., the join points with control flow in a dashed rounded line in the figure). The system also records which unit test cases match the sub-pointcut of each test-based pointcut.
2. At *program execution time* (the right hand side of the figure), whenever the target program reaches a new join point, the runtime system compares the current execution history against the ones recorded at the first stage. When it finds a match, there is a unit test case that produced the history. Then the pointcut whose sub-pointcut matched the unit test cases at testing time matches the current join point, hence the system runs the advice body.

4.2 Notification Aspect with Test-Based Pointcuts

Figure 9 shows a redefinition of the `updateRemoteDirectory` pointcut in the `ConnectionUpdate` aspect with test-based pointcuts. The unit test cases are also redefined like shown in Figure 10.

```
public class TestFtpConnectionChdir
        extends TestCase {
  FtpConnection con, errCon;
  ConnectionFixture f;
  ... //setUp and tearDown are omitted.
  public void testChdirSuccess() {
   f.changedDirName = "testDir";
   f.changedPath = HOME+"/testDir";
   Phase.beginTestCall();
   boolean r = con.chdir(f.changedDirName);
   Phase.endTestCall();
   assertTrue(r);
   assertEquals(f.changedPath,con.pwd());
  }
  public void testChdirFail() {
   f.unknownDirName = "unknownDir";
   f.unchangedPath = HOME;
   Phase.beginTestCall();
   boolean r = con.chdir(f.unknownDirName);
   Phase.endTestCall();
   assertFalse(r);
   assertEquals(f.unchangedPath,con.pwd());
  }
  public void testChdirFailWithNetworkError(){
   f.changedDirName = ""; f.changedPath = "";
   Phase.beginTestCall();
   boolean r = errCon.chdir(f.changedDirName);
   Phase.endTestCall();
   assertFalse(r);
   assertEquals(f.changedPath,errCon.pwd());
  }
}
```

Figure 10. The `TestFTPConnectionChdir` test case class revised for test-based pointcuts

The `updateRemoteDirectory` pointcut now matches join points that have the same execution histories as the unit test cases that reference the `changedDirName` field of the `ConnectionFixture` class. The second sub-pointcut `!within(SocketIO+)` ensures to exclude actions within subtypes of `SocketIO` from execution histories. This will be explained in Section 4.5.

The unit test cases in Figure 10 are different from the ones in Figure 7 in two points. First, they use the fields of the `ConnectionFixture` class, such as `changedDirName` or `unkownDirName`, instead of literal values. Second, they have two dummy method calls between phases of parameter setup, test execution, and validation. The calls to `Phase.beginTestCall()` and `Phase.endTestCall()` in Figure 10 separate test executions of the setup and validation operations, respectively.

Figure 11 shows the `ConnectionFixture` class, which defines fields that store the parameters and expected results in the unit test cases with respect to network connections.

The whole program runs in the following ways. At the testing time, the system executes all the unit test cases and finds that `testChdirSuccess` and `testChdirFailWithNetworkError` reference the field specified by the `updateRemoteDirectory` pointcut. At the same time, it records execution histories of the target method `chdir` called from each unit test case. For example, the execution history of `chdir` from `testChdirFailWithNetworkError` are the lines 2–7 in Figure 6[5]. At the program execution time, when an execution of `chdir` follows the same lines as the ones done by `testChdirFailWithNetworkError`, the pointcut matches, hence the advice runs.

[5] It also keeps track of the executions of methods called from `chdir`. We will discuss this in Section 4.4.

```
/** Fields in this class are for
  * storing testing values and expected
  * connection operations.
  */
public class ConnectionFixture {
  public String changedDirName;
  public String unknownDirName;
  ... //other fields follow here.
}
```

Figure 11. `ConnectionFixture` in the FTP client program

4.3 Advantages

We claim that the aspects defined with test-based pointcuts are more robust and more suitable to distinguish fine grained execution histories thanks to the following properties of unit test cases defined with fixtures.

- **Up-to-date**. When developers change a target program, they also change the unit test cases associated to the program so that all the unit test cases succeed. Therefore, test-based pointcuts automatically reflect the changes to the target program as long as the associated tests are maintained.
- **Thorough**. Good programs are supposed to have unit test cases that cover typical usages of each method. Therefore, we can expect that there already exist unit test cases for the target program, which are usable to distinguish particular behavior (i.e., execution history) of a method. Even if we had to define new unit test cases for defining test-based pointcuts, those unit test cases will improve quality of the program because they also serve as ordinary unit test cases.
- **Crosscutting classification**. Even though unit test cases are usually organized along with the structure of the target program, fixture fields help finding unit test cases related to a crosscutting concern. This is because a fixture field often related to a concern that crosscuts multiple unit test cases.

The aspect definition using test-based pointcuts solves the problems discussed in Section 2.2 and Section 2.3 in the following ways. When the target program evolves as shown in Figure 5, the developers will also add unit test cases for the `chdir` method of the newly introduced class `FilesystemConnection`. Since those new unit test cases also use the fixture fields `changedDirName` and `unkownDirName`, the pointcut in Figure 9 automatically matches join points in the `FilesystemConnection` class without modification.

The test-based pointcut in Figure 9 can distinguish differences of execution histories. As already explained, it can distinguish the three execution histories shown in Section 2.3. Moreover, it does not explicitly depend on the detailed implementation of `chdir` because the system automatically records histories of the test executions. In other words, we can say that test-based pointcuts provide abstractions of detailed method behavior so that aspects can distinguish different execution histories without directly relying on detailed implementations.

The fixture fields would help identifying crosscutting concerns that will appear upon future evolution. In the above example, the `changedDirName` fixture field represents changes of current directory name. Therefore, if a version of JFtp had another method that changes current directory, a unit test case for the method would use the same fixture field, which in turn helps the programmer to find relevance with the GUI updating concern.

4.4 Semantics

Below we explain the semantics of the `test` pointcut as processes at the testing and program execution times.

4.4.1 Testing Time

At testing time, the system compiles a target program without weaving aspects, and executes all unit test cases in order to identify specifying unit test cases and to record execution histories.

Given a pointcut description `test(`p`)`, the system identifies any unit test case that creates at least one join point matching the sub-pointcut p during the test execution. Note that p can be any pointcut but should not include a `test` pointcut. For example, given the pointcut in Figure 9:

```
test(get(* ConnectionFixture.changedDirName))
```

the system runs all the unit test cases in Figure 10, and finds that `testChdirSuccess` and `testChdirFailWithNetworkError` match the subpointcut.

When the system executes each unit test case, it also records the execution history between calls to `Phase.beginTestCall()` and `Phase.endTestCall()`. As for the unit test cases in Figure 10, the system records histories of the executions of the `chdir` method of the `FtpConnection` class. A later section will describe the definition of an execution history.

4.4.2 Pointcut Matching at Program Execution Time

At program execution time, `test(`p`)` matches a method execution join point when the execution history of the join point is the same as the history of one of the unit test cases that matched p. Since an execution history of a method execution is only available at the end of the execution, our system compares execution histories only when a test-based pointcut is used with an `after` advice declaration. When used with a `before` or `around` advice declaration, the pointcut unconditionally matches execution join points of target methods that were executed by a unit test case matching p.

The following subsections explain the definition of an execution history and inclusion between execution histories.

4.4.3 Execution Histories in the Control Flow Graph

We define an execution history of a method with respect to the control flow graph (CFG) of the method. An execution history is a sequence of CFG nodes. A node of a CFG of a method is a segment of bytecode instructions in the target method that corresponds to a *join point shadow* [16, 20]. We extended the definition of the join point shadow to include any bytecode instruction, such as branches and accesses to local variables. We call the extended join point shadows *fine-grained join point shadow*s (FJPSs). Join points generated from extended FJPSs are only used for comparing execution histories. The semantics of existing pointcut primitives is thus unchanged.

A CFG edge is a directed relation between two FJPSs and individually labeled. We write an edge e from node a to b as $e = \langle a, b \rangle$.

There are two kinds of edges in the CFG, namely *regular* and *irregular*. An irregular edge denotes a transition caused by an exception. A regular one does any other transition including a conditional jump and a method call.

Figure 12 shows an example of a CFG constructed from the `chdir` method in Figure 6. For readability, we omit several nodes. Each circle labeled with ni denotes a CFG node, which is associated with a (virtual) bytecode instruction on its right. A solid arrow between two nodes is a CFG edge. A dashed arrow denotes that we omitted some nodes and edges between two nodes of the CFG. For example, we omit eight sequentially connected nodes between `n2`

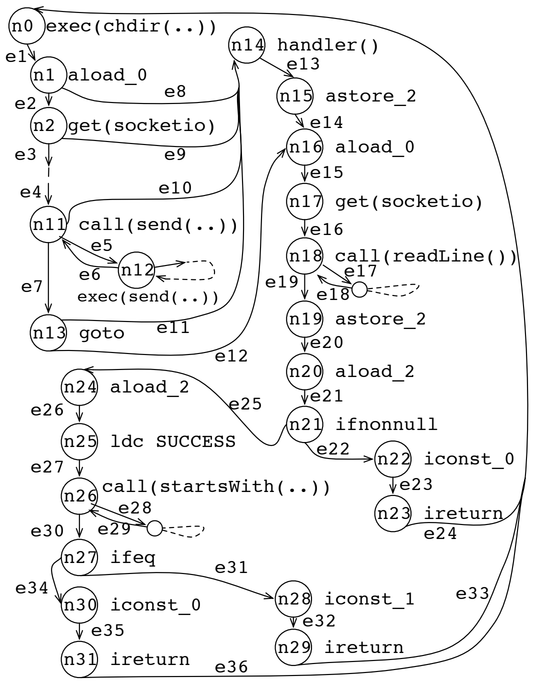


Figure 12. CFG of the `chdir` method

and `n11`. Compared with standard instruction-level CFGs, ours are different in the following respects:

- In addition to real bytecode instructions, we introduced virtual bytecode instructions in order to represent a code block that corresponds to an AspectJ join point. In the example, `n0` and `n12` are virtual instruction nodes that correspond to execution join points.
- We integrate intra-procedural and inter-procedural edges into one CFG. Therefore, call nodes (`n11`, `n18` and `n26`) are connected to their callee nodes as well as their subsequent instruction nodes. When a virtual method call can dispatch to more than one implementations, there will be edges between the caller node and each execution node.
- There is an irregular edge from every instruction node to each exception handler node (if exists), or to the execution node of the method (if there is no handler). In the example, `e8`, `e9`, `e10` and `e11` are irregular edges. There are also irregular edges from nodes `n15` through `n31` to `n0`, which are omitted in the Figure 12.

4.4.4 Execution History

An execution history of a method execution is a set of CFG nodes whose elements are instruction nodes executed during the execution. When `chdir` is executed and fails due to a network error, its execution history is {`n1`, `n2`, $\cdots$, `n11`, `n12`, `n14`, `n15`, `n16`, `n17`, `n18`, `n19`, `n20`, `n21`, `n22`, `n23`}.

4.4.5 Matching Execution Histories

We define that an execution history h_e of method m at execution time matches a history h_t at the testing time if and only if h_e contains all instruction nodes in h_t; i.e., $h_t \subseteq h_e$.

Since we represent an execution history by using a set of instruction nodes, there are cases when two different execution histories are determined as the same. One is different behavior over iteration. The above definition of execution history abstracts actual execution histories by discarding the number of iterations because it is based on the sets of CFG nodes. Assume two executions of a statement `while(···) {if(···)` A `else` B `}`. The histories under our definition are equal if the one execution performs A then B, and the other does B then A. Another case is because when we need to distinguish edges that lead to a node, especially in case of exception handling. We believe that future study would reveal appropriate representations of execution histories.

4.5 Filtering Execution History

As already mentioned (Section 4.1), there is an optional syntax of test-based pointcuts that is useful to ignore differences between testing and execution environments. When a test-based pointcut has an optional sub-pointcut, it specifies to filter out edges from execution histories that do not match the sub-pointcut.

For example, a pointcut

```
test(get(* ConnectionFixture.changedDirName),
    !within(SocketIOStub))
```

has the same meaning as the version without the filter except for ignoring operations in the `SocketIOStub` class. This hides differences between `SocketIO` objects used at testing time and runtime. The unit test cases use so-called mock objects that mimic network connection because it is not feasible to establish a network connection at the testing time. With the mock objects, the execution histories at the testing time and at the runtime are completely different in `send` and `readLine`.

This filtering mechanism is also useful to exclude operations that do not substantially change behavior of a target method such as logging operations.

5. Implementation

We implemented a prototype compiler that supports test-based pointcuts by extending the AspectBench Compiler (abc) [5] [6]. The size of the extension is approximately 3800 lines of Java code.

This section explains how our implementation realized the semantics presented in the previous section. The main challenges are:

1. an overall compilation process that first records execution histories, and then generates efficient executables, and
2. an efficient implementation of pointcut matching at runtime.

5.1 Overall Compilation Process

In order to pre-record execution histories of unit test cases, a compilation of test-based pointcuts needs a process into a current AspectJ's compilation strategy. Before explaining our compilation process, we first review the compilation strategy of AspectJ.

5.1.1 Compilation Strategy of AspectJ Programs

AspectJ's compilation strategy [5, 16] consists of the following three steps:

1. Per-module compilation. Given source files of class declarations and aspect declarations, the compiler compiles each of them into an intermediate representation.
2. Processing inter-type declarations. An inter-type declaration is a static crosscutting mechanism that can add method, field or other declarations into existing classes from aspects. The compiler modifies the intermediate representation of classes according to the inter-type declarations.
3. Advice weaving. Given an advice declaration with a pointcut, the compiler looks for join point shadows that can match the pointcut at runtime. For each of such shadows, it inserts a sequence of instructions that calls the body of the advice. The instructions include conditional branches if the pointcut requires runtime tests (e.g., `if` and `args` pointcut primitives).

After those steps, the compiler generates a class file from the intermediate representation. Note that inter-type declarations should be woven before the advice declarations because inter-type declarations can change class structures, which could affect pointcut matching.

5.1.2 Recording Execution Histories

Before weaving advice declaration, our compiler records execution histories of all the unit test cases. It is achieved by generating compiled code instrumented with the following two kinds of code fragments.

1. History recording code. In order to record execution histories, the compiler inserts recording code into entry of every basic block in all methods in the target program. A basic block is a sequence of nodes, when viewed in a CFG, each of which has only one outgoing regular edge except for the last one. The inserted code records if the test run executed the code at least once, when the corresponding FJPS matches the filtering pointcut. By only recording at the entry of each basic block, the implementation reduces the size of execution histories.
2. Sub-pointcut-matching code. The compiler also inserts a code fragment into each join point shadow that matches the sub-pointcut of a `test` pointcut. It records unit test cases matching per a `test` pointcut basis.

After running each unit test case, the compiler obtains (1) an execution history of the unit test case, and (2) a set of `test` pointcuts whose sub-pointcut matched the test case.

5.2 Efficient Pointcut Matching

Our implementation optimizes pointcut matching by eliminating obvious overheads that a naive implementation would have. First, our set-based semantics of an execution history enables more efficient implementation than a sequence-based semantics. In addition, our implementation avoids obvious overheads by:

(1) only monitoring code blocks that are executed by unit test cases,

(2) only checking at entries of basic blocks, and

(3) using bit vectors for recording flags.

Even though there is room for further optimizations such as the ones used in path profiling techniques [6], our current implementation exhibits, as an initial step, reasonable performance as we will see in Section 6.1.

Below, we explain how our compiler instruments code by taking a case when `chdir` in Figure 6 is compiled with `ConnectionUpdate` in Figure 9 and `TestFtpConnection` in Figure 10.

The first compilation stage finds that the sub-pointcut in the pointcut matches two unit test cases, namely `testChdirSuccess` and `testChdirFailWithNetworkError`. Let's assume that the compiler gives indices 0 and 1, respectively to those two execution histories. It also records respective execution histories of those unit test cases. The compiler prepares sets of flags to respective execution histories in order to monitoring code blocks executed at runtime. The runtime system sets one of the flags at the entry of each basic block in CFG that are visited by an execution of a unit test case. At the end of each target method, the runtime system checks all the expected flags in order to compare execution

[6] The prototype implementation is publicly available at `http://www.graco.c.u-tokyo.ac.jp/ppp/projects/testbasedpc`.

```
 1  public boolean chdir(String n) {
 2    TraceCFlow $1 = ConnectionUpdate.$trace$1;
 3    $1.push(2); $1.set(0,3); $1.set(1,3);
 4    try { $1.add(0,2); $1.add(1,2);
 5     try {
 6      socketio.send("CWD " + n);
 7     } catch (Exception e) { $1.add(1,1); }
 8     String l = socketio.readLine();
 9     boolean $r;
10     if (l == null) { $1.add(1,0);
11      $r = false;
12     } else { $1.add(0,0);
13      if (l.startsWith(SUCCESS)) { $1.add(0,1);
14       $r = true;
15      } else {
16       $r = false;
17     } }
18     return $r;
19    } finally {
20     if ($1.check())
21      ConnectionUpdate.aspectOf().after$1();
22     $1.pop();
23  } }
```

Figure 13. The `chdir` method after instrumentation (manually decompiled)

Table 1. Methods of `TraceCFlow`

`push(`n`)`	creates an array of bit vectors for n unit test cases
`set(`t`,`b`)`	allocates b-bit vector for unit test case t
`add(`t`,`b`)`	sets a flag for basic block b of unit test case t
`check()`	returns `true` if all flags of any unit test case are set
`pop()`	merges top two arrays of bit vectors into one

histories. In the case of `chdir`, the compiler prepares flags for nodes `n0`, `n24` and `n28` with respect to `testChdirSuccess`, and `n0`, `n14` and `n22` with `testChdirFailWithNetworkError`.

The runtime system manages sets of flags for each advice declaration, by using the `TraceCFlow` class, which is a thread-local stack of arrays of bit vectors. The methods of `TraceCFlow` are summarized in Table 1.

Figure 13 shows (decompiled) code generated by the compiler. At the beginning of the method, it obtains a stack by obtaining a `TraceCFlow` object (line 2), pushes an array of bit vectors and allocates bit vectors (line 3). For each entry of basic block that was executed by a unit test case (e.g., lines 4 and 7), it sets flags in respective bit vectors. The code that runs an advice body (line 21) is guarded by checking if all flags of any unit test case are set (line 20). Before returning from the method, it merges the current bit vectors into the ones under the stack top (if there exists). This is because, when the method is recursively called, the execution history of the inner call should be part of the one of the outer call as well.

One of the potential problems of our optimization is impreciseness at exception handling. Since we computed basic blocks by only concerning regular edges, a set of flags does not indicate how many instructions in a basic block are executed when an exception is thrown. For example, when an exception is thrown during an execution of the following code, we cannot determine either statement throws the exception: `try {` A `;` B `} catch(`$\cdots$`) {`$\cdots$`}`.

Table 2. Average execution times (μsecs)

	simplified			
	NA	**AJ**	**TB**	**TB/AJ**
Success	6.38	6.43	14.6	2.27
Failure	5.30	5.41	13.1	2.42
NetworkError	26.0	26.1	34.6	1.33

	original			
	NA	**AJ**	**TB**	**TB/AJ**
Success	43.7	43.7	84.0	1.92
Failure	36.5	37.4	69.6	1.86
NetworkError	72.1	74.2	112	1.51

6. Evaluation

We evaluated test-based pointcuts with the respect to runtime performance (Section 6.1), feasibility of test-based pointcuts (Section 6.2), and robustness against evolution (Section 6.3). Section 6.2 and 6.3 mainly focus on rewriting AspectJ pointcuts into test-based ones. Due to lack of practical examples, we have not performed quantitative evaluation of pointcuts that require to distinguish fine-grained behavior. This is left for future study.

6.1 Preliminary Performance Measurement

In order to confirm feasibility of our implementation, we measured and compared execution times of several micro-benchmark programs compiled with either an aspect with current AspectJ pointcuts or the one with test-based pointcuts. We executed all measurement on an Intel Core Duo 2.16 GHz with 2 MB L2 cache memory, 2 GB RAM and a HotSpot JVM version 1.5.0_07 under Mac OS X 10.4.10.

We executed each program for five times and chose the shortest overall execution time. The benchmark programs are:

- a *simplified* `chdir` implementation presented in Section 5.2, and
- the *original* `chdir` implementation in JFtp version 1.48 consisting of 11 methods or 180 lines of code. We slightly modified the code to support unit testing.

All programs are executed with a mock object that simulates network communications.

These programs are compiled with the following three aspect configurations: (**NA**) without aspects, (**AJ**) with a current AspectJ aspect which manually monitors a calls to `getLine` inside `chdir` implementation, and (**TB**) with an aspect using test-based pointcuts. These aspects simply increment an integer filed in their advice declarations. The lengths of allocated bit vectors for those aspects are 8 (in *simplified*) and 49 (in *original*).

We measured execution times of each configuration of each `chdir` implementation with the following three runtime parameters: by averaging the elapsed time of a loop repeated for ten million times.

- (*Success*) a correct directory name and a functional network connection,
- (*Failure*) a non existing directory name and a functional network connection, and
- (*NetworkError*) a correct directory name and a broken network connection.

Table 2 shows the execution times. As the **TB/AJ** columns show, **TB** configurations are slower than **AJ**s by the factors of 1.33 to 2.42. Note that the overhead factors are those of the worst cases since we measured the execution times of the advised method only,

Table 3. Number of pointcut definitions converted into test-based pointcuts. (The first three applications are from the AspectJ distribution, and the last four are from AspectJ in Action [24].)

		FC					
Application	**SC**	Total	1	2	3	**Test Cases**	**Fixture Fields**
observer	1	0	0	0	0	1	1
telecom	5	0	0	0	0	14	6
spacewar	19	3	0	3	0	39	45
factorial	1	1	1	0	0	3	2
account	2	0	0	0	0	4	5
auth	3	5	2	0	3	6	6
transaction	3	10	2	1	7	7	7
Total	34	19	5	4	10	74	72

which does not include network communication time. We believe the overheads are reasonable as an initial implementation, though it still has room for further optimization.

6.2 Feasibility of Test-based Pointcuts

Clearly, not all pointcuts in AspectJ can be rewritten into test-based pointcuts. In order to see how oftenly we can use test-based pointcuts, we rewrote pointcuts in existing AspectJ applications into test-based pointcuts. We took seven non-trivial programs from the AspectJ distribution[7] and a text book [24]. Since they come without unit test cases, we defined unit test cases by ourselves. Table 3 summarizes the results of the experiment. Each column shows:

- (**SC**) the number of the test-based pointcuts that are successfully converted,
- (**FC**) the number of the pointcuts that cannot be converted into test-based pointcuts,
- (**Test Cases**) the number of the unit test cases that we defined along with the test-based pointcuts, and
- (**Fixture Fields**) the numbers of the fixture fields that are declared for the unit test cases.

Approximately three fifth of pointcuts can also be expressed by test-based pointcuts. Other 19 pointcuts that are converted can be classified to the following groups.

1. Compositions of named pointcuts, such as `factorialOperation(n) && !cflowbelow(factorialOperation(int))`. Even though it might be possible to convert such a pointcut into test-based one, there are no points to do so because the pointcut is sufficiently abstract.
2. Universally matching pointcuts, such as `call(* *.*(..))`. It is difficult to select unit test cases universally due to lack of common fixture variables.
3. Pointcuts that capture library access. Even though we can define unit test cases for library accesses, we did not do so as it is unlikely to test libraries from application programs, and library APIs are ususally stable and do not cause fragile pointcut problems.

6.3 Robustness against Evolution

In order to evaluate how test-based pointcuts make can aspects more robust against software evolution, we compared two AOP implementations of open source software systems over several versions.

[7] `http://www.eclipse.org/aspectj/`

6.3.1 Setup

We chose two open source software systems written in Java, selected several versions in their archives, and refactored several crosscutting concerns into two kinds of aspects, namely current AspectJ aspects and aspects with test-based pointcuts. We also wrote unit test cases for the latter, when necessary.

The chosen software systems and versions are follows:

1. JFtp network file browser versions *1.07*, *1.15* and *1.48*. We refactored a *GUI updating concern* that refreshes screen upon changes in a remote directory.
2. Scarab issue tracking system[8] versions *b16*, *b19*, *b20* and *b21*. It is a medium size web application having hundred thousands lines of code. We refactored (1) a *security concern* that checks user's permissions[9], (2) an *email concern* that reports actions through e-mail, and (3) a *caching concern* that reuse database objects.

6.3.2 Counting Mismatch

The current AspectJ aspects do not use wildcard in the pointcuts. This is because, from the original implementation, it was hard to derive pointcuts with wildcards that accurately reflect developers' intention. Therefore, the evaluation should be taken as the maximal improvements by using test-based pointcuts from a naive aspect implementation.

For each version of the AO refactored implementations, we applied the aspects to the classes in the next version, and counted the number of methods in the next version that are accidentally advised and the number of methods that are not advised while they should be.

6.3.3 Results

Table 4 shows the number of mismatches. We separately discuss the cache concern in Scarab in Section 6.3.4. Each row in the table corresponds to one version, indicating the following numbers.

- (**Mismatches**) The numbers of join point shadows that either should be advised in the current version but missed by the pointcuts in the previous version, or should not be advised but captured by the previous pointcuts. Therefore, lower the numbers are, more robust pointcuts are. The numbers t/a correspond to the number of mismatches by test-based pointcuts and current AspectJ pointcuts, respectively. Following the total numbers of mismatches, the numbers in **CC**, **MA**, **MD** and **SC** columns show the breakdown by the causes of mismatches, which are explained below.

[8] `http://scarab.tigris.org/`

[9] We refactored security concerns that involve with 8 permission types among 19 in Scarab.

Table 4. Number of mismatching join point shadows captured by pointcuts in previous versions

		Mismatches						
	Version	**Total**	**CC**	**MA**	**MD**	**SC**	**UC**	**JPS**
GUI updating concern	*1.07*						1/0	10
in JFtp	*1.15*	6/10	6/6	0/4	-	-	12/0	8
	1.48	3/24	3/3	0/21	-	-	-	32
security concern	*b16*							13
in Scarab	*b19*	-	-	-	-	-		13
	b20	0/1	-	0/1	-	-		14
	b21	4/8	2/2	2/4	(1)	0/1		17
email concern	*b16*							18
in Scarab	*b19*	1/1	1/1	-	-	-		19
	b20	-	-	-	-	-		19
	b21	0/2	-	0/1	(1)	-		19

- (**UC**) The number of unintended captures. An unintended capture is a join point shadow that is not advised in one version but matches pointcut in the previous version. For example, in a Java version of JFtp, there are two methods that implement the same functionality yet have different number of parameters. For some reasons, only one of them updates the GUI. In the test-based pointcut version, both methods are advised due to the almost same unit test cases defined for them. As we cannot find good reasons to explain this inconsistency in the original Java version, we excluded those shadows from mismatches[10].
- (**JPS**) The number of join point shadows (i.e., target methods) that are advised by aspects.

We classified the versions of the mismatches into the following categories and analyzed the effectiveness of test-based pointcuts for each categories:

- *Concern Change* (**CC**); when a developer decided to apply aspects to different join points from the previous version. We have to modify pointcuts by all means in this case.
- *Method Addition* (**MA**); when a new version has a newly defined method that should be advised by an aspect. The test-based pointcuts can help as long as the new method also has proper unit test cases. We found introduction of sibling classes (e.g., `FilesystemConnection` in JFtp as discussed in Section 2.2), or introduction of a new operation into a group of operations for a class (e.g., in the methods the `ModifyIssue` class in Scarab) are typical cases of this category. Since those newly defined methods are similar to existing ones, defining unit test cases that use shared fixture variables is not so difficult.
- *Method Deletion* (**MD**); when a method that is advised in the previous version does not exist in the new version. Both current AspectJ and test-based pointcuts do not match in the new version in this case. Since both have no problems upon evolution we merely indicated the number of deleted methods in parenthsis in the table.
- *Signature Change* (**SC**); when the signature of a method changes from the previous version. The test-based pointcuts can automatically match the new version as long as unit test cases are properly modified. This is a reasonable assumption because compilation (or execution) of unit test cases easily identifies the cases that are not compatible with the modified method definitions. Using wildcards in current AspectJ aspect would improve robustness as well, but not always useful especially to unanticipated changes.

[10] In Scarab, we did not count the number of unintended captures. However, we conjecture to be zero as it has minimal number of unit test cases, and it is more consistent than JFtp.

6.3.4 Caching Concern in Scarab

It turned out difficult to define test-based pointcuts for the caching concern in Scarab. This is because, Scarab applies a caching mechanism to, not all of, but some of the database accesses, presumably selected by efficiency and safety criteria. Therefore, it is not possible for test-based pointcuts to conjecture such properties from parameters in the unit test cases. An explicit collection of unit test cases for database accesses that should be cached could help little, when compared against current AspectJ pointcuts that enumerate all the method signatures.

6.3.5 Analysis of the Result

Through the case study, we found that the test-based pointcuts can improve robustness against evolution especially when signature of a method changes, and when a new member (e.g., a method or a class) is introduced into a family.

On the other hand, we also found that test-based pointcuts may not be suitable to crosscutting concerns that are selectively applied based on non-functional properties such as efficiency.

7. Related Work

7.1 Techniques to Cope with Fragile Pointcut Problem

To address the fragile pointcut problem, several studies proposed techniques or programming conventions that prevent or detect aspects that depend on details of implementations. For example, XPI [12] and Open Modules [1, 21] restrict aspects so that they will only depend on the interface that is explicitly provided. Those proposals certainly make aspects robust, but make it difficult to precisely apply aspects.

Another approach to robust pointcut is to use high-level program information instead of signatures in detailed implementations. The approach includes semantics- or analysis-based pointcuts, such as ALPHA [22] and CARMA [13] and SCoPE [3]. Even though defining accurate rules are not easy, we believe this approach would compliment ours.

Similarly, there are approaches that exploit information at earlier development stages. For example, model-based pointcuts [17] can use information in design models. Motorola WEAVER [8] can use information in behavioral specifications. Both systems require mechanisms to map high-level information to the code, which should be provided by developers. Test-based pointcuts use unit test cases, which are already associated to the code in many cases.

Rather than automatically solving the fragile pointcut problem, there are tools that detect it. Such tools include the pointcuts delta analysis [26] and JMantlet [7]. The test-based pointcuts can also detect problems by running unit test cases, but also can resolve the problems by modifying the test cases.

7.2 More Precise Aspect Application

There are studies that make aspects sensitive to execution history. Those include tracecuts [27], tracematches [2], the work by Douence et al. [9, 10, 11] and ALPHA [22]. Most prominent difference from our proposal is whether a pointcut description depends on implementation details of a target program. Since pointcuts in those studies specify execution histories based on a language of join points, individual points have to be specified by means of traditional pointcuts. The test-based pointcuts, on the other hand, specify execution histories by using examples of executions, i.e., unit test cases, which do not need to specify detailed behavior.

There are attempts to introduce finer grained join points, such as loop continuations [15] and conditional branches [23], into AspectJ like languages. The test-based pointcuts can also distinguish branches, without directly relying on implementation details.

8. Conclusion

This paper proposed test-based pointcuts as an extension to AspectJ. A test-based pointcut indirectly matches method execution join points through fixture fields and unit test cases. We believe those indirection steps nicely abstract crosscutting concerns. Fixture fields, or test parameters can capture representative values specific to a concern. Unit test cases can abstract a specific execution history. With the help of automated test execution tools, those indirect steps can be validated against the target program.

Those properties make test-based pointcuts a novel solution that addresses both robustness and precision of pointcuts. Of course, test-based pointcuts do not automatically solve the fragile pointcut problem, because the unit test cases must be consistent with the target program, which usually has to be done by developers. In this sense, our proposal merely shifted the maintenance cost from pointcuts to unit test cases, the latter of which has to be paid anyway in practical software development processes.

We implemented a compiler that supports test-based pointcuts on top of the AspectBench compiler. The implementation executes unit test cases at compilation time and records execution histories. At the program execution time, instrumented compiled code efficiently checks similarity of execution histories. Our initial implementation showed an overhead factor of merely 1.33–2.42 in a kernel method.

We also carried out a case study that evaluates how much test-based pointcuts can alleviate the fragile pointcut problem in practical open-source software systems. The study suggests that test-based pointcuts are useful to program evolution due to signature changes and method additions into a family of classes or methods.

There are many interesting challenges left for future work. The algorithm and implementation of the mechanism in that record and compare execution histories should be improved. Precision of execution history is another interresting issue. While we define a history as a set of executed basic blocks in a CFG, alternative ones that can distinguish edges might be better. At the same time, it would be useful to have a mechanism that can specify unrelevant parts in the history.

Acknowledgements

This work is partly supported by a JSPS Research Fellowship for Young Scientists.

We would like to thank the members of the PPP research group and the TM-seminar at University of Tokyo for their valuable comments, especially Jan Hanneman, Tomoyuki Aotani and Kazunori Kawauchi. We also thank the reviewers and the participants of LATE'07 at which we presented an earlier version of the paper [25], Mira Mezini, Klaus Ostermann and their colleagues for their insightful comments, and the anonymous reviewers of the AOSD conference for valuable feedback on the paper.

References

[1] J. Aldrich. Open modules: modular reasoning about advice. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 144–168, July 2005.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 345–364, 2005.

[3] T. Aotani and H. Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 161–172, 2007.

[4] T. Aotani and H. Masuhara. Towards a type system for detecting never-matching pointcut compositions. In *FOAL '07: Proceedings of the 6th workshop on Foundations of Aspect-Oriented Languages*, pages 23–26, 2007.

[5] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98, 2005.

[6] T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[7] J.-S. Boulanger and M. P. Robillard. Managing concern interfaces. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 14–23, Washington, DC, USA, 2006.

[8] T. Cottenier, A. van den Berg, and T. Elrad. Joinpoint inference from behavioral specification to implementation. In *ECOOP '07: Proceedings of the 21th European Conference on Object-Oriented Programming*, pages 476–500, 2007.

[9] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 173–188, London, UK, 2002.

[10] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 141–150, 2004.

[11] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with Arachne. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 27–38, 2005.

[12] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

[13] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-oriented software development*, pages 60–69, 2003.

[14] S. Hanenberg, D. Stein, and R. Unland. From aspect-oriented design to aspect-oriented programs: tool-supported translation of JPDDs into code. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 49–62, 2007.

[15] B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 63–74, 2006.

[16] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, 2004.

[17] A. Kellens, K. Mems, J. Brichau, and K. Gybels. Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. In *ECOOP '06: Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 501–525, 2006.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, 2001.

[19] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[20] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC '03: Proceedings of the 12th International Conference Compiler Construction 2003*, pages 46–60, 2003.

[21] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding open modules to AspectJ. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 39–50, 2006.

[22] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 214–240, 2005.

[23] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 181–191, 2005.

[24] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications, 2003.

[25] K. Sakurai and H. Masuhara. Test-based pointcuts: a robust pointcut mechanism based on unit test cases for software evolution. In *LATE '07: Proceedings of the workshop on Linking Aspect Technology and Evolution revisited*, March 2007.

[26] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of Aspect-oriented software. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 653–656, 2005.

[27] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 159–169, 2004.