to appear in Proceedings of APLAS'03 - the First Asian Symposium on Programming Languages and Systems, Beijing, November 2003

Dataflow Pointcut in Aspect-Oriented Programming

Hidehiko Masuhara and Kazunori Kawauchi

Graduate School of Arts and Sciences, University of Tokyo
Tokyo 153–8902 Japan
{masuhara,kazu}@graco.c.u-tokyo.ac.jp

Abstract. A dataflow-based pointcut is proposed for aspect-oriented programming (AOP) languages. The pointcut specifies where aspects should be applied based on the origins of values. It is designed to be compatible with the other kinds of pointcuts in existing AOP languages. Primary application fields of the pointcut are the aspects in which flow of information is important, such as security. This paper presents the design of the pointcut with a web-application example, and its prototype implementation.

1 Introduction

Aspect-oriented programming (AOP) languages support modularization of crosscutting concerns[6, 11]. A concern crosscuts a software system when its implementation does not fit in existing modules such as procedures and classes, and causes modifications to many modules without AOP. Previous studies have shown that AOP supports modularization of such crosscutting concerns as profiling[5], persistence[16], distribution[9] and optimization[3, 17].

One of the important mechanisms in AOP is called pointcut-and-advice, which is found in many AOP languages including AspectJ[10]. The mechanism allows the programmer to declare additional or alternative behaviors to program execution by writing aspects. An aspect is a module that contains a set of advice declarations. A join point is a point in execution whose behavior can be affected by advice. Each advice declaration has a pointcut that specifies a set of join points and a body that specifies behavior at the specified join points.

1.1 Pointcut-and-Advice Mechanism in AspectJ

For concreteness of the discussion, the rest of the paper assumes the AspectJ's pointcut-and-advice mechanism. However, most part of the discussion should also be applicable to the other AOP languages with similar mechanisms.

In AspectJ, the join points are operations on objects, including method calls, object constructions and accesses to instance variables. The pointcuts are written in a sub-language that has primitive predicates on the kinds of join points, binders to extract values from join points, and operators to combine pointcuts.

An advice declaration consists of a keyword to specify the timing of execution, a pointcut to specify matching join points, and a block of Java statements to specify behavior.

The following is an example aspect, which adds a "cookie" to every generated response in Servlet[4]-based web applications:

```
aspect CookieInsertion {
  pointcut generatePage(ServletResponse r) :
    call(void Servlet+.do*(..)) && args(*,r);
  after(ServletResponse r): generatePage(r) {
    r.addCookie(new Cookie("...","..."));
  }
}
```

The second and third lines define a pointcut named `generatePage`, which matches method call join points with certain signatures, and extracts a value from the join point. The inside of the `call` pointcut specifies that the method must be defined in a subclass of `Servlet` (N.B. "`+`" means any subclass) and the name of the method begins with `do` (N.B. "`*`" means any string). The `args(*,r)` pointcut binds variable `r` to the second argument of the call so that the advice body can access to it.

The third to fifth lines are an advice declaration. The keyword `after` means that the execution of the advice follows the execution of a matching join point. After the keyword, there are a formal parameter to the advice (in the parentheses), a pointcut, and a body of the advice (in the curly brackets). To sum up, when `doPost` method (or `doGet`, etc.) finishes, the advice calls `addCookie` method on the second parameter to `doPost` method.

1.2 Expressiveness of Pointcuts

In pointcut-and-advice mechanisms, pointcuts serve as the interface between an aspect and a target program. In order to apply aspects to many but particular points in a program, elaborated pointcut descriptions are needed. It is sometimes difficult to write pointcuts when the pointcut language does not have appropriate predicates that distinguish the characteristics of the application points.

The claim of the paper is that a pointcut that can address the origins of data, or *dataflow* between join points, is useful to write concise aspects. This is especially so for security-related aspects, in which flow of information is primary considerations. To the best of the author's knowledge, there have been no such pointcuts in existing AOP languages.

We propose a new kind of pointcut, called `dflow` pointcut, that identifies join points based on the origins of data. It is designed as an extension to AspectJ's pointcut language; `dflow` pointcuts can be used in conjunction with the other kinds of pointcuts in AspectJ. This makes it easy for the programmers to adopt `dflow` with minimal efforts.

The rest of the paper is organized as follows. Section 2 gives an example problem. Section 3 presents the design of the dataflow-based pointcut, and how it

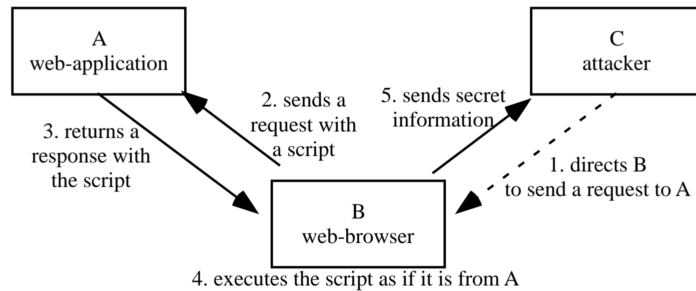

Fig. 1. An Exploit of a Cross-site Scripting Problem

can solve the problem. Section 4 presents a prototype implementation to clarify the semantics of the pointcut. Section 5 shows approaches to efficient implementations. Section 6 discusses related work. Section 7 summarizes the paper.

2 Motivating Example: Fixing a Security Problem in Web-Applications

2.1 Cross-site Scripting Problem

Cross-site scripting is a security problem in web-applications. By exploiting a flaw of a web-application, an attacker can reveal secret information from the browser of the web-application's client[1]. The following scenario with three principals, namely (A) a web-application, (B) a web-browser of a client of A, and (C) an attacker, explains an attack (Fig. 1):

1. C directs B to send a request to A[1]. The request contains a malicious script as a parameter.
2. B sends the request to A. For example, the request may go to a login page of a shopping site, and a script is embedded as a parameter to the ID field.
3. A returns a web page as a response to B's request. A has a cross-site scripting problem when the malicious script appears as a part of the response. For example, the shopping site may return a page that indicates a failure of login with the given ID.
4. B executes the malicious script in the returned page with privileges for A.
5. The malicious script accesses secret information in B that should only be accessed by A.

The problem could have been avoided if A did not return the malicious script to B. A solution to this problem on the A's side is not to generate pages by using a string that comes from any untrusted principal. A simple implementation is to remove special characters that constitute scripts from strings that come

[1] This can be accomplished by giving B a document with a link to a malicious address, or by exploiting a web-browser's bug.

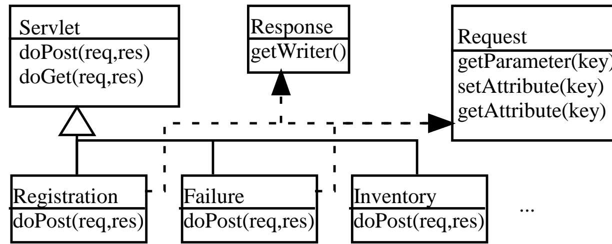

Fig. 2. Structure of a Web Application

from untrusted principals, and to replace them with non-special (or quoted) characters. This solution is usually called *sanitizing.*

We here define a following sanitizing task for a web-application:

> Assume that the web-application is to generate a web page[2]. When a string that originates from anyone untrusted by the web-application appears in the generated page, it replaces special characters in the string with quoted ones.

By replacing special characters with quoted ones, the browser that receives a generated page no longer interprets the string as an executable script.

2.2 How Sanitizing Crosscuts a Web Application

In the web-applications that dynamically generates many pages, the sanitizing can be a crosscutting concern because its implementation could involve with many parts of the program. Here, we assume a web-application on Java servlets framework[4], which defines a class for each kind of pages. Fig. 2 shows a class structure of the web-application. Each `Servlet` subclass represents a particular kind of pages. When a client browser sends a request for a URL, the framework automatically creates an instance of a respective `Servlet` subclass, and then invokes `doPost` (or `doGet`, etc.) with objects representing the request and a response. The method can read values to the input fields from the request object, and generates a new page by writing into a stream in the response object. Alternatively, a method can delegate another method to generate a page after performing some process.

Fig. 3 shows definitions of some of the `Servlet` subclasses. When `doPost` method of `Registration` class is invoked, it merely checks whether the requested ID is found in the database, and transfer itself to either `Inventory` page or `Failure` page. In `Failure` class, it displays a message of failure with the

[2] Another approach is to replace special characters when the web-application receives strings from browsers. It is not recommended because the replacement might affect the intermediate process unexpectedly. Also, it can not sanitize strings that come to the application via methods other than HTTP[2].

```
class Registration extends Servlet {
  void doPost(Request req, Response res) {
    String id = req.getParameter("ID");       //read input field
    if (<id is found in the database>)
      <transfer to an Inventory page>
    else {
      req.setAttribute("PREV", req.getURL()); //store current URL as an
      <transfer to a Failure page>            //originating address of the
    }                                         //transferred page
  }
}

class Failure extends Servlet {
  void doPost(Request req, Response res) {
    PrintWriter out = res.getWriter();
    out.println("<HTML>...Login failed for: ");
    out.println(req.getParameter("ID"));      //read input field & print
    out.println("...<a href=");
    out.println(req.getAttribute("PREV"));    //back to the originating page
    out.println(">go back</a>...");           //by using the stored address
  }
}
```

Fig. 3. Implementation of `Servlet` Subclasses
(Type names are abbreviated due to space restrictions.)

requested ID. It also places a link to the previous page by reading information in the attributes of the request.

The sanitizing task is to wrap the call to `getParameter` method in `Failure` class with a method that replaces special characters. Although the task needs to change only one place in the application, it crosscuts the application because the similar modifications need to be done in many sibling classes when those classes also use the results from `getParameter` for responding.

This paper assumes that `getParameter` is the only source of interested strings. This assumption can be easily relaxed by enumerating the other sources as well as `getParameter` in the pointcut definitions in the rest of the paper.

2.3 Usefulness of Dataflow

Since the sanitizing task crosscuts, it looks a good idea to implement it as aspects. However, it sometimes is not easy to do so with existing pointcut-and-advice mechanisms because they do not offer pointcuts to address dataflow, which is the primary factor in the sanitizing task.

The problem can be illustrated by examining the (incomplete) aspect definition in Fig. 4. The `respondClientString` pointcut is supposed to intercept any join point that prints an unauthorized string to a client. (By unauthorized, we

```
aspect Sanitizing {
  pointcut respondClientString(String s) : ...; // incomplete

  Object around(String s) : respondClientString(s) {
    return proceed(quote(s));           //continue with sanitized s
  }

  String quote(String s) {
    return <replace all special characters in s>;
  }
}
```

Fig. 4. (Incomplete) Aspect for the Sanitizing Task

mean that a string is created from one of client's input parameters.) With properly defined `respondClientString`, the task of sanitizing is merely to replace all special characters in the unauthorized string, and then continue the intercepted join point with the replaced string[3].

With existing kinds of pointcuts, it is not possible to write an appropriate `respondClientString` in a declarative manner.

- First, a straightforward definition is not appropriate. For example, the following pointcut will intercept strings that are to be printed as a part of a response to a client. However, it will intercept strings including 'authorized' ones:

```
pointcut respondClientString(String s) :
 call(* PrintWriter.print*(String)) && args(s)
 && within(Servlet+);
```

- Second, even if one could write an appropriate pointcut definition, the definition is less declarative. For example, an appropriate `respondClientString` could be defined with auxiliary advice declarations that detect and trace unauthorized strings. However, those advice declarations are not declarative because they have to monitor every operations that involve with (possibly) unauthorized strings.

In order to define `respondClientString` in a declarative manner, a pointcut that can identify join points based on the origins, or *dataflow*, of values is useful. The next two reasons support this claim.

First, dataflow can use non-local information for determining the points of sanitizing. For example, the result of `getAttribute` in `Failure` class in Fig. 3 needs no sanitizing because if we trace the dataflow, it turns out to be originally obtained by `getURL` (i.e., not by `getParameter`) in `Registration` class.

[3] In AspectJ, `proceed` is a special form to do so. When the formal parameters of the advice (i.e., `s`) is bound by `args` pointcut (e.g., `args(s)`), the arguments to the `proceed` (i.e., `quote(s)`) replace the original arguments in the continued execution.

Second, dataflow can capture derived values. For example, assume that a web-application takes a parameter string out from a client's request, and creates a new string by appending some prefix to the string, the new string will also be treated as well as the original parameter string.

3 Dataflow Pointcut

We propose a new kind of pointcut based on dataflow, namely `dflow`. This section first presents its syntax and example usage for the simplest case, and then explains additional constructs for more complicated cases.

3.1 Pointcut Definition

The following syntax defines a pointcut p:

$$\begin{aligned} p &::= \texttt{call(}s\texttt{)} \mid \texttt{args(}x\texttt{,}x\texttt{,...)} \mid p\texttt{\&\&}p \mid p\texttt{||}p \\ &::= \texttt{dflow[}x\texttt{,}x\texttt{](}p\texttt{)} \mid \texttt{returns(}x\texttt{)} \end{aligned}$$

where s ranges over method signature patterns and x ranges over variables.

The first line defines existing pointcuts in AspectJ[4]. `call(`s`)` matches calls to methods with matching signatures to s. `args(`x_1`,...,`x_n`)` matches any calls to n-arguments methods, and binds the arguments to the respective variables $x_1, \ldots, x_n$. Operators `&&` and `||` combine pointcuts conjunctively and disjunctively, respectively.

The second line defines new pointcuts. `dflow[`x`,`x'`](`p`)` matches if there is a dataflow from x' to x. Variable x should be bound to a value in the current join point. (Therefore, `dflow` must be used in conjunction with some other pointcut, such as `args(`x`)`, that binds x to a value in the current join point.) Variable x' should be bound to a value in a past join point matching to p. (Therefore, p must have a sub-pointcut that binds a value to x'.) `returns(`x`)` is similar to `args`, but binds a return value from the join point to variable x. This is intended to be used only in the body of `dflow`, as a return value is not yet available when a current join point is created.

By using `dflow`, the pointcut for the sanitizing task can be defined as follows:

```
pointcut respondClientString(String o) :
  call(* PrintWriter.print*(String)) && args(o) && within(Servlet+)
  && dflow[o,i](call(String Request.getParameter(String))
              && returns(i));
```

The second line is not changed from the one in Section 2.3, which matches calls to print methods in `Servlet` subclasses, and binds the parameter string to variable `o`. The `dflow` pointcut restricts the join points to such ones that the parameter string originates from a return value of `getParameter` in a past join point.

[4] AspectJ has more pointcuts than the ones presented here. We omit them for simplicity.

3.2 Dataflow Relation

The condition how `dflow` pointcut identifies join points can be elaborated as follows: assume x is bound to a value in the current join point, `dflow[`x`,`x'`](`p`)` matches the join point if there exists a past join point that matches p, and the value of x originates from a value bound to x' in the past join point. By originating from a value, we mean the value is used for deriving an interested value. For example, when two strings are concatinated, the concatinated string originates from those two strings. We let the originating-from relation be transitive; e.g., the origins of the two strings are considered as the origins of the concatinated string as well.

The originating-from relation is defined as follows. Let v and w are two values. v originates from w when

- v and w are the identical value, or
- v is a result of a primitive computation using $u_1, \ldots, u_n$, and u_i originates from w for some i $(1 \leq i \leq n)$.

The above definition is sufficient for languages only with primitive values. When a language also has compound values, such as arrays and objects, we need to extend the matching condition for `dflow`. Although it needs further study to give a feasible condition, we tentatively extended the definition in the following ways. `dflow[`x`,`x'`](`p`)` matches when the value of x or a value reachable from the value of x originates from the value of x' or a value reachable from the value of x'.

3.3 Excluding Condition

We also defined an extended syntax of `dflow` for excluding particular dataflows. We call the mechanism `bypassing`.

A motivation of `bypassing` can be explained in terms of the sanitizing task. Assume a `Servlet` subclass that manually quotes the client's inputs:

```
class ShippingConfirmation extends Servlet {
  void doPost(Request request, Response response) {
    PrintWriter out = response.getWriter();
    String address = quote(request.getParameter("ADDR"));
    out.print("...Please confirm delivery address:");
    out.print(address);
    ...
  }
}
```

When an object of this class runs with `Sanitizing` aspect after filling its pointcut definition with the one in Section 3.1, the aspect intercepts method calls that print `address` in `ShippingConfirmation`. As `address` has already quoted string, it doubly applies `quote` to the quoted string.

The following extended `dflow` syntax excludes dataflows that go through certain join points:

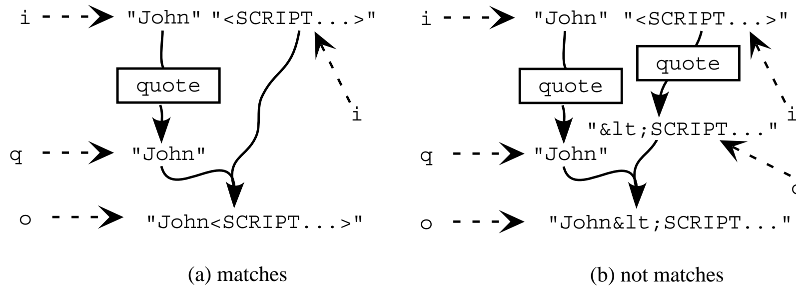

Fig. 5. How dataflow is restricted in `dflow[o,i](...) bypassing[q](...)`.

$$p ::= \mathtt{dflow}[x,x](p)\ \mathtt{bypassing}[x](p)$$

Intuitively, a `bypassing` clause specifies join points that should not appear along with a dataflow. By using `bypassing`, the following pointcut avoids intercepting quoted strings:

```
pointcut respondClientString(String o) :
  call(* PrintWriter.print*(String)) && args(o) && within(Servlet+)
  && dflow[o,i](call(String Request.getParameter(String))
              && returns(i))
     bypassing[q](call(String *.quote(String)) && returns(q));
```

The `bypassing` clause requires that `o` (an argument to print method) to originate from `i` (a return value from `getParameter`) but not through `q` (a return value from `quote`) after `i`.

Precisely, `bypassing` requires existence of at least one dataflow that does not go through join points matching to the pointcut in the `bypassing` clause. Fig.5 illustrates computations that generate concatenated strings from two strings. Assume that the original strings at the top of the figure are the results of `getParameter` in the above example. Then the `dflow` pointcut with `bypassing` clause matches the computation (a) because there is a dataflow to the string at the bottom of the figure without going through `quote`. On the other hand, it does not match the computation (b) because all dataflows go through `quote`; i.e., there are no dataflows bypassing `quote`.

The semantics of `bypassing` clause can be defined by slightly extending the originating-from relation. Let v and w are two values. v originates from w bypassing x in p, when:

- there have been no such a join point that matches p and the value bound to x is identical to v, and
- either of the following conditions holds:
  - v and w are the identical value, or
  - v is a result of a primitive computation using $u_1, \ldots, u_n$, and u_i originates from w bypassing x in p for some i $(1 \leq i \leq n)$.

3.4 Explicit Dataflow Propagation

We provide an additional declaration form that specifies explicit propagation of dataflow through executions in external programs. This is useful in an open-ended environment, where a program runs with external code whose source programs are not available (e.g., a class library distributed in a binary format).

A declaration is written as a member of an aspect in the following form:

`declare propagate:` p `from` x,x,... `to` x,x,...;

where p and x range over pointcuts and variables. The form requests that, when a join point matching to p is executed, it will regard that the values of the `to`-variables originate from the values of the `from`-variables.

For example, assume that a program uses `update` method of `Cipher` class for encryption (or decryption), but the system has no access to the source code of the class. With the following declaration, the system will regard that the return value from `Cipher.update` originates from its argument. As a result, if a string matches `dflow` pointcut, the `Cipher` encrypted string of the string also matches to the `dflow` pointcut.

```
aspect PropagateOverEncryption {
  declare propagate: call(byte[] Cipher.update(byte[]))
        && args(in) && returns(out) from in to out;
}
```

The propagate declarations are designed to be reusable; i.e., once someone defined propagate declarations for a library, the users of the library merely need to import those declarations to track dataflow over the library.

The propagate declarations would be sufficient for the libraries that have only dataflows between input and output parameters. Coping with more complicated cases, such as the ones involving with structured data or the ones with conditional dataflows, is left for further study.

3.5 Notes on Scalability

The proposed pointcut has potential for more realistic and complicated situations than those in the examples presented so far. Below, we discuss two possible situations.

More Sources of Unauthorized Strings. A web-application would obtain unauthorized values by executing methods other than `getParameter`. For example, results of `Request.getURL()` might contain an arbitrary string. Or, a web-application may manually unquote strings for some purposes.

The sanitizing aspect can cope with such situations by enumerating possible sources of unauthorized values in the `dflow` pointcut. The following definition is a revised pointcut:

```
pointcut respondClientString(String o) :
  call(* PrintWriter.print*(String)) && args(o) && within(Servlet+)
  && dflow[o,i]((call(String Request.getParameter(String))
                 || call(String Request.getURL())
                 || call(String *.unquote(String)))
                && returns(i));
```

where the fourth and fifth lines are for incorporating additional sources.

External Storage. Practical web-applications often use external storage for persistently keeping information. Since `dflow` pointcuts (or any other information flow analysis techniques) only keep track of the dataflows in an program, an unauthorized value stored in external storage could be recognized as authorized after it gets retrived.

We believe that AOP helps to solve such problems by allowing the programmer to extend the sanitizing aspect that manually traces unauthorized values.

- It has an additional advice declaration to mark unauthorized data in a database. When the web-application writes an unauthorized value into a database, the advice puts a flag into an extra attribute of the written record. Such an advice can be defined in a similar manner to the existing sanitizing advice.
- The extended `respondClientString` pointcut captures values that originate from a value in the database but only when the value has a flag. This extension can be achieved by adding a pointcut to match reads of the database with conditional expression to check the flag into the `dflow` sub-pointcut.

4 Prototype Implementation

We build a prototype implementation[5] of a language with `dflow` in order to clarify the semantics, and to establish a standpoint for optimized implementations. It is implemented as an extension to Aspect SandBox (ASB)[14], which is a modeling framework for AOP mechanisms including the pointcut-and-advice.

The prototype implementation is based on a simple, pure object-oriented language with a pointctu-and-advice mechanism simplified from the one in AspectJ. It dynamically tests and executes pointcuts and advice; i.e., when a join point is to be executed (e.g., a method is to be called), it looks for advice declarations that have pointcuts matching to the join point, and executes the advice bodies. The detailed semantics and mechanism can be found in the other literatures[14, 23].

The implementation for `dflow` consists of the three parts: (1) keeping track of dataflow of values, (2) matching a join point to a `dflow` pointcut, and (3) dealing with `propagate` declaration. The following three subsections explain those in this order.

[5] Available from `http://www.graco.c.u-tokyo.ac.jp/ppp/projects/dflow/` .

For simplicity, the language lacks the following features: primitive values, exceptions, dynamic class loading, static type system, and wildcards in pointcuts.

4.1 Dataflow Tags

In order to track a dataflow, the system gives a unique identifier to each `dflow` pointcut. We hereafter call such an identifier a dataflow tag. The system also associates a mutable cell to each value for recording a set of dataflow tags. A notation $\texttt{dflow}^t\texttt{[}x\texttt{,}y\texttt{](}p\texttt{)}$ expresses a pointcut $\texttt{dflow[}x\texttt{,}y\texttt{](}p\texttt{)}$ with its tag t. The following three operations manipulate dataflow tags of a value v: $get(v)$ retrives a set of associated tags from v, $add(v, T)$ (where T is a set of dataflow tags) puts all tags in T to the associated tags of v, and $remove(v, T)$ takes all tags in T from the associated tags of v. We also write $get(x)$, $add(x, T)$, and $remove(x, T)$ for denoting operations on the value bound to variable x.

The dataflow tags associated with a value represent the following property: for $\texttt{dflow}^t\texttt{[}x\texttt{,}y\texttt{](}p\texttt{)}$, when $t \in get(v)$, there was a join point that matches p and v originates from the value of y in p. This property is preserved in the following ways:

1. It associates an empty dataflow tags to newly created value v.
2. When a join point is created, for each pointcut

   $$\texttt{dflow}^t\texttt{[}x\texttt{,}y\texttt{](}p\texttt{)}\ \texttt{bypassing[}z\texttt{](}q\texttt{)}$$

   in a program, it matches the join point to p and q, respectively. When the join point matches p, it performs $add(y, \{t\})$. When the join point matches q, it performs $remove(z, \{t\})$.[6] (For a pointcut without `bypassing` clause, q is regarded as a pointcut that matches no join points.)
3. When primitive operation $o(v_1, \ldots, v_n)$ yields a result value v_r, it performs $add(v_r, get(v_1) \cup \cdots \cup get(v_n))$.

4.2 Pointcut Matching

It is straightforward to match a join point to `dflow` pointcut. Pointcut $\texttt{dflow}^t\texttt{[}x\texttt{,}y\texttt{](}p\texttt{)}$ matches when $t \in get(x)$.

Note that $\texttt{dflow[}x\texttt{,}y\texttt{](}p\texttt{)}$ pointcut must be used in conjunction with another pointcut (e.g., $\texttt{args(}x\texttt{)}$) that binds a value to variable x. In order to match sub-pointcuts in a right order, a compound pointcut is first transformed into its disjunctive normal form. Then, for each conjunctive pointcut $p_1 \texttt{\&\&} \cdots \texttt{\&\&} p_n$, all p_is that do not have `dflow` are first matched, which bind values to variables. The remaining p_is with `dflow` test dataflow tags of the values bound to the variables. This rule is basically the same to `if` pointcut in AspectJ, which can test values of variables bound by the other pointcut.

[6] Note that *add* and *remove* operations are side-effecting in order to cope with aliasing.

4.3 Explicit Dataflow Propagation

A `propagate` declaration causes a propagation of dataflow tags at the matching join points. Given the following declaration:

$$\texttt{declare propagate: } p \texttt{ from } x_1\texttt{,}\ldots\texttt{,}x_m \texttt{ to } y_1\texttt{,}\ldots\texttt{,}y_n\texttt{;}$$

when a join point matches p, it performs $add(y_i, get(x_1) \cup \cdots \cup get(x_m))$ for all i $(1 \leq i \leq n)$.

Note that the system should manipulate dataflow tags after the execution of the join point if a variable is bound by `returns` pointcut. This should not be a problem because the advice mechanism has no access during the execution of the external code.

5 Towards Efficient Implementation

A compiler-based implementation is being designed. The implementation (1) statically matches as much pointcuts as possible, and (2) eliminates unnecessary dataflow tag operations. Our approaches are to (1) use AspectJ compiler after translating `dflow` pointcuts into existing pointcuts and advice, and to (2) use static dataflow analysis.

5.1 Translating `dflow` into AspectJ

Pointcut matching, which is dynamically performed in the prototype implementation, can statically matched to the locations in the source program in most cases[15].

The first step naively translates `dflow` pointcuts into AspectJ's `pointcuts` and advice. The AspectJ compiler will then eliminates the overheads of dynamic pointcut matching for the existing kinds of pointcuts.

The implementation has to associate dataflow tags for each value. A set of dataflow tags can be represented as bit-vectors since we have fixed number of `dflow` declarations in a given program. For regular objects, the bit-vectors can be installed by using inter-type declaration mechanism in AspectJ or by extending Java virtual machine. For primitive values and arrays, we translate the given program into the one that explicitly manage such bit-vectors separately.

Management of dataflow tags can be implemented as advice declarations in AspectJ. Basically, $\texttt{dflow}^t\texttt{[}x\texttt{,}y\texttt{](}p\texttt{)}$ pointcut is translated into `if` pointcut an advice declaration. The pointcut itself becomes the following pointcut that checks dataflow tags of the specified variable:

$$\texttt{if (}\langle t \text{ is in } get(x)\rangle\texttt{)}$$

Additionally, the pointcut generates the following advice declaration for associating a dataflow tag to a value:

$$\texttt{before(}y\texttt{) : } p \texttt{ \{ } \langle \text{perform } add(y, \{t\})\rangle \texttt{ \}}$$

(or it may be `after` advice when y is bound by `returns` pointcut in p). In addition, for each primitive operation, a code fragment that propagates dataflow tags will be inserted. This should be done by program transformation as primitive operations do not create join points in AspectJ[7].

AspectJ compiler statically resolves most pointcut matchings. The remaining overheads in the translated program are (1) the `if` pointcuts to test dataflow tags (but this is not significant as they appear at limited locations, and each test merely takes constant time), (2) the advice to associate dataflow tags (also, this is not significant as they merely sets a bit at limited locations), and (3) the operations to propagate dataflow tags upon primitive operations.

5.2 Eliminating Unnecessary Dataflow Tags

The second step eliminates dataflow tags that are proven to be useless. Each pointcut p has the source code locations that can create matching join points. Given a $\texttt{dflow}^t\texttt{[}x\texttt{,}y\texttt{](}p\texttt{)}$ pointcut, it can determine the source code locations where dataflows from y can reach. Existing information flow analysis should be able to do this job. Then, from the source code locations where no dataflows of `dflow` pointcut can reach, we can eliminate dataflow tag operations. Similary, when static analysis finds that a class involves with no dataflow, the slots for dataflow tags can be eliminated from the class.

Even at locations where some dataflow reach of, dataflow tag operations can be optimized by aggregating them per basic block (i.e., a series of operations that has no control transfers). For example, the following code, which computes the magnitude of two complex numbers, will have an dataflow tag operation for each primitive computation:

```
r0=r1*r2-i1*i2; i0=r1*i2+r2*i1; m=Math.sqrt(r0*r0+i0*i0);
```

However, those operations can be aggregated as

$$add(\texttt{m}, get(\texttt{r1}) \cup get(\texttt{i1}) \cup get(\texttt{r2}) \cup get(\texttt{i2}))\ .$$

Those optimizations will ensure that programs without `dflow` pointcut have no overheads. The authors expect that the optimizations will eliminate a large amount of overheads from programs with `dflow`, but the quantitative evaluations are yet to be done.

6 Related Work

Giving more expressiveness to pointcuts in AOP languages are studied in many ways. Some offer pointcuts that can examine calling context[10], execution history[21], and static structures of a program[7].

[7] In an AOP language that offers more flexible selection of join points (e.g., [20]), it would be possible to write advice declarations that propagates dataflow tags over primtive operations.

Demeter is an AOP system that can declaratively specify traversals over object graphs[12, 13]. It allows to examine relation between objects, but the relation is about a structure of data in a snapshot of an execution.

There are systems that can examine dataflow in a program either in a static or dynamic manner (e.g., Confined Types[18] and taint-checks in Perl[22]). Those are useful for *checking* security enforcement. On the other hand, when a breach of the security enforcement is found by those systems, the programmer may have to fix many modules in the program without AOP support.

Information flow analyses (e.g., [19]) can detect a secret that can leak by indirect means, such as the conditional context and timing. For example, the following code does not have direct dataflow from `b` to `x`, but information about `b` indirectly leaks in `x`:

```
if (b) { x = 0; } else { x = 1; }
```

As we have shortly discussed, our dataflow definition only deals with direct information flow. It does not regard a dataflow from `b` to `x`. Extending dataflow definition to include such indirect information flow, is left for future study.

7 Conclusion

We proposed `dflow` pointcut that identifies join points based on the dataflow of values in aspect-oriented programming (AOP) languages. The pointcut allows the programmers to write more declarative aspects where the origins of data are important. The pointcut is designed to be used with the other existing kinds of pointcuts. An interpreter-based implementation is developed on top of Aspect SandBox.

The usefulness of the `dflow` pointcut is explained by an example—sanitizing web-applications. Although the paper showed only one example, we believe the notion of dataflow would be useful in many situations. Our plan is to apply the pointcut to many programs, such as design patterns[8] and the other kinds of security problems.

The design space of the `dflow` pointcut is large enough for further study. Especially, to find a right balance between the declarativness of the pointcut and the runtime efficiency is curcially important. It will also be crucially important to give a formal framework of `dflow` pointcut so that we can reason about completeness of the semantics.

Acknowledgments

The authors would like to thank Naoki Kobayashi and Atsushi Igarashi for their comments on an early draft of the paper. The authors would also like to anonymous reviewers for their comments.

References

1. CERT. Malicious HTML tags embedded in client web requests. Advisory Report CA-2000-02, CERT, February 2000.
2. CERT Coordination Center. Understanding malicious content mitigation for web developers. Tech tips, CERT, February 2000.
3. Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT symposium on Foundations of software engineering*, pages 88–98, Vienna, Austria, 2001.
4. James Duncan Davidson and Suzanne Ahmed. Java servlet API specification: Version 2.1a. Technical Document of Sun Microsystems, November 1998. http://java.sun.com/products/servlet/.
5. Jonathan Davies, Nick Huismans, Rory Slaney, Sian Whiting, Matthew Webster, and Robert Berry. Aspect oriented profiler. a practitioner report presented at AOSD2003, March 2003.
6. Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Comm. ACM*, 44(10):29–32, October 2001.
7. Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD2003)*, pages 60–69. ACM Press, 2003.
8. Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA2002)*, pages 161–173, November 2002.
9. Mik A. Kersten and Gail C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In *Proc. ACM Conf. Object-oriented Programming, Systems, Languages, and Applications*, pages 340–352. ACM, 1999.
10. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 2001.
11. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.
12. Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Comm. ACM*, 44(10):39–41, October 2001.
13. Karl J. Lieberherr. *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996.
14. Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In Luca Cardelli, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP2003)*, volume 2743 of *Lecture Notes in Computer Science*, pages 2–28, Darmstadt, Germany, July 2003. Springer-Verlag.
15. Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of 12th International Conference on Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60, 2003.

16. Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD2003)*, pages 120–129. ACM Press, March 2003.
17. Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia L. Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD2003)*, pages 110–119. ACM Press, 2003.
18. Jan Vitek and Boris Bokowski. Confined types. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA99)*, pages 82–96. ACM Press, 1999.
19. D. Volpano, G. Smith, and C Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
20. David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ICFP2003*, 2003.
21. Robert J. Walker and Gail C. Murphy. Implicit context: Easing software evolution and reuse. In *Proceedings of the eighth international symposium on Foundations of software engineering for twenty-first century applications (FSE-8)*, volume 25(6) of *ACM SIGSOFT Software Engineering Notes*, pages 69–78, San Diego, California, USA, November 2000.
22. Larry Wall and Randal Schwartz. *Programming Perl*. O'Reilly and Associates, 1991.
23. Mitchell Wand, Gregor Kiczales, and Chris Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In Ron Cytron and Gary T. Leavens, editors, *Foundations of Aspect-Oriented Langauges (FOAL2002)*, Technical Report TR#02–06, Department of Computer Science, Iowa State University, pages 1–8, Enschede, The Netherlands, April 2002.