HAL open science

Vidock: A Tool for Impact Analysis of Aspect Weaving on Test Cases

Romain Delamare, Freddy Munoz, Benoit Baudry, Yves Traon

To cite this version:

Romain Delamare, Freddy Munoz, Benoit Baudry, Yves Traon. Vidock: A Tool for Impact Analysis of Aspect Weaving on Test Cases. 22nd IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS), Nov 2010, Natal, Brazil. pp.250-265, 10.1007/978-3-642-16573-3_18 . inria-00555069v2

HAL Id: inria-00555069

https://inria.hal.science/inria-00555069v2

Submitted on 12 Aug 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Vidock: a Tool for Impact Analysis of Aspect Weaving on Test Cases

Romain Delamare[1], Freddy Munoz[1], Benoit Baudry[1], and Yves Le Traon[2]

[1] IRISA / INRIA Rennes {rdelamar,fmunoz,bbaudry}@irisa.fr
[2] LASSY, University of Luxembourg yves.letraon@uni.lu

Abstract. The addition of a cross-cutting concern in a program, through aspect weaving, has an impact on its existing behaviors. If test cases exist for the program, it is necessary to identify the subset of test cases that trigger the behavior impacted by the aspect. This subset serve to check that interactions between aspects and the base program do not introduce some unexpected behavior. Vidock performs a static analysis when aspects are compiled with a program to select the test cases impacted by the aspects. It leverages the pointcut descriptor to locate the set of methods impacted by aspects and then selects the test cases that can reach an impacted method. This static analysis has to perform over-approximations when the actual point where the aspect is executed can be computed only at runtime and when test cases call polymorphic objects. We measure the occurrence of these assumptions in 4986 projects containing 498 aspects to show they have a limited impact. Then, we run experiments with Vidock on 5 cases studies and analyze the impacts that different types of aspects can have on test cases.

1 Introduction

Aspect-oriented Programming (AOP) [1] encapsulates crosscutting behaviors into single units called *aspects*. The intent of this programming paradigm is to improve the readability, modularity and maintainability of the code. An aspect is composed of an *advice*, which realizes the cross-cutting behavior, and a *pointcut descriptor* (PCD), which describes the points in the program where the advice is woven (called joinpoints).

The cross-cutting concern either keeps the base program's behavior untouched by adding a new behavior, or might modify the existing behavior. An aspect can replace a routine execution when needed, or access the value of a protected data structure at runtime. Thus, the introduction of aspects can modify the control or the data flow of the program in which it is inserted.

When an aspect is woven in a program it is necessary to ensure that it interacts correctly with the program. If the program has already been tested, this implies identifying the test cases that are impacted by the aspect weaving to run them to check the integration of the aspect in the program. A test case is considered impacted if it covers a joinpoint matched by the aspect.

We propose an analysis which does not require executing any test case. The Vidock tool is based on a static analysis performed in two steps. First, the

analysis leverages the pointcut descriptor that identifies all the points in the program that are impacted by the aspect weaving. Then, the test cases and the base program are analyzed to determine the points reached by the test cases. If a test case reaches a point that is impacted by an aspect, it is identified as an impacted test case. The advantage of such a static analysis is that the analysis is less time consuming than solutions relying on dynamic analysis.

Because our analysis is static, it has to perform over-approximations on impacted points in two cases. First, a pointcut can declare a dynamic expression. In that case, we statically over-approximate the set of joinpoints that can be matched by the aspect. Second, we have to over-approximate the coverage of test cases in case of a method call on a polymorphic object, since it is not possible to statically know the type of the object. The benefit of these over-approximations is that our impact analysis identifies all the test cases that are impacted. We chose to build Vidock for AspectJ aspects and JUnit test cases.

In section 2 we recall the main constructs in AOP. Section 3, presents the details of the analysis and section 4, discusses the hypotheses for this work: the choice of an impact analysis for programs that compile without the aspects and the over-approximation for dynamic pointcuts and in the presence of polymorphism in the base code. This discussion is based on empirical measurements on 46 open source Java projects that use aspects. Section 5 describes Vidock that implements this automatic analysis as a plug-in for the Eclipse platform. Section 6 presents experiments that apply Vidock on various programs. In section 7 we discuss related work.

2 Illustrating example

Aspect-Oriented Programming (AOP) is a programming paradigm that separates cross-cutting concerns. AOP encapsulates concerns that crosscut across several modules into *aspects*. In our work, we used AspectJ, a popular and mature implementation of AOP for Java. In this section we present the core concepts of AspectJ through a running example.

2.1 Auction system

We illustrate AOP through examples extracted from an online auction system developed using Java and AspectJ. A user can sell an item by creating an auction with an end date and a minimum price. Other users can place bids on this auction until it closes. The user who placed the highest bid (if any) wins the auction. The system insures that the seller will be paid: a bidder must credit his account before he can bid, and the system checks that the total of all his bid is not greater than the amount available on his account. When the auction closes, money is immediately transferred from the buyer's account to the seller's account.

Two aspects have been added to the base program. The first aspect, Reserve, gives a seller the option to set a secret reserve price. If the highest bid does not match the reserve price then the item is not sold. The second aspect, AltBid,

```
1  public privileged aspect Reserve {
2    private int Auction.reservePrice = 0;
3    pointcut closeOpen(Auction a):
4      execution(void Open.close(Auction));
5    void around(Auction auction): closeOpen(auction) {
6      if(auction.reservePrice>0) { ... }
7      else proceed(auction);
8    }
9  }
```

Listing 1.1. The Reserve aspect.

```
1  public aspect AltBid {
2    pointcut getAmount(): call(int Bid.getAmount())
3      && cflow(execution(* *.close(Auction)))
4    int around(): getAmount() {
5      ...
6      int res = secondBid.getAmount()*11/10;
7      return res;
8    }
9  }
```

Listing 1.2. The AltBid aspect.

changes the way the final price is computed. In the base program the price an item is sold is the amount of the highest bid. With this aspect, the price is the amount of the second highest bid plus 10%. Listings 1.1, 1.2 respectively show an excerpt of the Reserve and AltBid aspects.

2.2 Aspect Oriented Programming: The case of AspectJ

AOP encapsulates crosscutting behaviors into single units called *aspects*. An aspect itself is composed of several units realizing the crosscutting behavior, these units are called *advices*. Aspects also provide pointing elements that designate well defined points in the program execution or structure where the crosscutting behavior is executed. The pointers are called *pointcut descriptors* (PCD) and the execution points *joinpoints*.

In AspectJ, a PCD is defined as a combination of names and keywords. Names are used to match specific places in the base program. For instance, the name `void Open.close(Auction)` in listing 1.1 (line 4) matches the method `close` which returns type `void`, receives a parameter of type `Auction` and is declared in the class `Open`. Names can contain wild-cards that enlarge the number of matches. The wild-cards `*` in the name `* *.close(Auction)` of listing 1.2 (line 3) are used to specify that the method can be declared in any class (`* *.close`) and have any return type (`* *.close`). Keywords define when the places matched by names will be intercepted. For instances, in the PCD `execution(* *.close(Auction))` the keyword `execution` indicates that the execution of the matched places (method `close(Auction)`) will be captured. The combination of names and keywords is referred as *expression*. PCDs

can be composed of multiple expression joint by the logic operators `&&` (conjunction) and `||` (disjunction). For instances the PCD in listing 1.2 (lines 2-3) is the conjunction of two expressions.

PCDs can also point joinpoints that occur dynamically during the execution of the program. For instance, the PCD of listing 1.2 (lines 2-3) is composed of two expressions. The first (line 2) intercepts the calls to `getAmount()`. The second (line 3) constrains the interception to the calls occurring inside the control flow of the execution of `close(Auction)`. To know whether a call to `getAmount()` occurs during the execution of `close(Auction)`, the program must be running. Therefore, there is no mean to statically know the exact occurrence of these joinpoints. We refer to these joinpoint as *dynamic joinpoints* and to a PCD pointing these points a *dynamic-PCD*.

AspectJ extends the Java syntax to allow developers to implement advices as natural as possible. Therefore, advices can be seen as routines that are executed at some point. Typically AspectJ advices are bounded to a PCD designating the points where they will be executed. The advice in listing 1.1 (lines 5-8) is bounded to the PCD `closeOpen`, therefore, it will be executed during the execution of `close(Auction)`. AspectJ provides three different kind of advices *Before*, *After* and *Around* indicating the moment when they are executed. Before and After advices are executed respectively just before / after reaching the joinpoints designated by the PCD. Around advices are a special type of advice, they are executed instead of the designated joinpoints. Besides, by invoking the special operation `proceed` they can execute the captured joinpoint. For instance, the advice in listing 1.1 (lines 5-8) executes the captured joinpoint only given a special condition (line 7), otherwise it replaces its execution (line 6).

3 Selection of the impacted test cases

To determine which test cases of the base program are impacted by the aspects, we statically analyze the Java base program, the aspects and the test cases. We analyze the PCD to select the set of methods impacted by an aspect. Then we can statically analyze the test cases to determine if they are impacted.

Figure 1 presents an overview of the analysis process. Initially, Vidock gathers information from the AJDT and the Spoon [2] tools. Then, driven by each test case description, it constructs a static call graph (SCG) – one for each test case (a,b). Besides Vidock selects all the methods having at least an aspect weaved, namely impacted methods (c). Finally, Vidock calculates the impacted test cases by intercepting the impacted methods with the SCG nodes (d).

3.1 Selecting methods

The first part of our analysis deals with the selection of the methods impacted by the aspect weaving. The method selection is divided in two stages, (1) we gather a set of joinpoints pointed by a PCD (using AJDT) and then (2) we calculate the set of methods related to those joinpoints (using Vidock). However, the selection

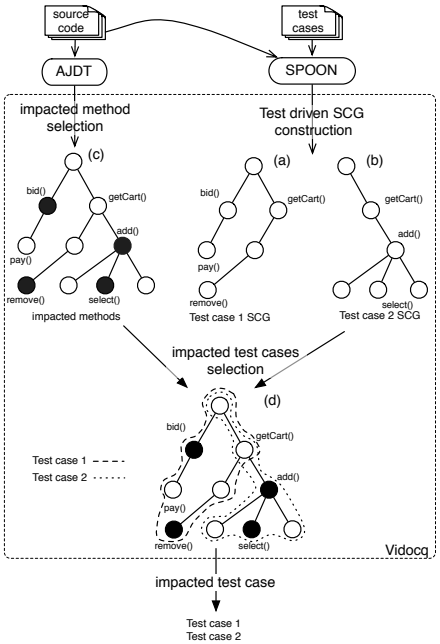

Fig. 1. Process overview.

of the methods *impacted* by aspects is not straightforward. The first issue we must deal with is the presence of dynamic-PCDs. Because dynamic-PCDs cannot be computed statically, it is not feasible to know the set of methods they point.

Expressions pointing dynamic joinpoints (with keyword such as `if`, `cflow` and `cflowbelow`) are used to constrain the amount of joinpoints pointed by static expressions. To deal with dynamic-PCDs statically, AJDT performs an over-approximation of the selected joinpoints. Such over-approximation consists in removing the dynamic expressions from the PCD. The resulting PCD points a set of joinpoints that contains those pointed by its dynamic version.

Formally, this over-approximation can be described as follows. Let P be a program, C be the set of all the PCDs declared in P and J be the set of all the joinpoints in a program P. Let $f_{pc} : C \rightarrow \mathcal{P}(J)$ be a function that returns the set of all the joinpoints pointed by a PCD. Let $f_{over} : C \rightarrow \mathcal{P}(J)$ be an over-approximation function. Given a PCD, it removes its dynamic part (if any) and gives the set of all the joinpoints it points.

$$\forall c \in C, f_{pc}(c) \subseteq f_{over}(c)$$

In the best case scenario, C contains no dynamic-PCDs and then $f_{over} = f_{pc}$.

Figure 2 illustrates this over-approximation. In the figure, the dynamic-PCD `getAmount()` (A) points the set of joinpoints $f_{pc}(A) = J_d$. However, it is unfeasible to compute J_d statically. The static version of `getAmount()` (B) points a larger set of joinpoints $f_{pc}(B) = J_s$ that contains those pointed by A. AJDT

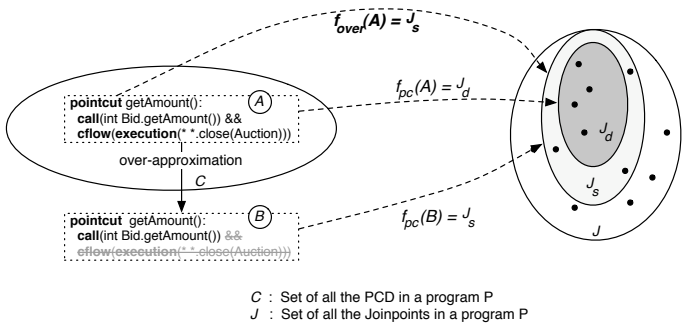

Fig. 2. Dynamic-PCD pointed joinpoints over-aproximation.

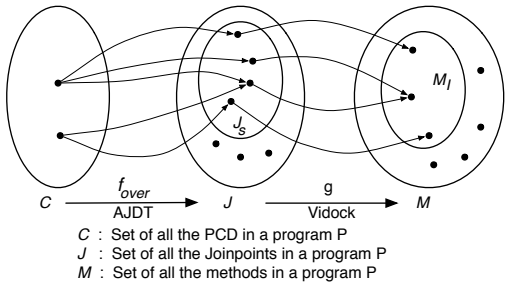

Fig. 3. Schematic view of the definition of an *impacted method*.

performs this over-approximation directly on the dynamic-PCD. This results in a set of joinpoints $f_{over}(A) = J_s$, which contains J_d and is also computable.

The second issue we must deal with is the actual selection of the *impacted* methods. An impacted method is any method related to a joinpoint that is pointed by a PCD. It can be specified as follows.

Let M be the set of all the methods in a program P. Let $g : J \rightarrow M$ be a function that, for a given joinpoint, gives the method where it is located.

$$g(j) = \begin{cases} \text{the caller} & \text{if } j \text{ is a call joinpoint} \\ \text{the executed method} & \text{if } j \text{ is an execution joinpoint} \\ \text{the constructor} & \text{if } j \text{ is an initialization joinpoint} \\ \text{the accessor method} & \text{if } j \text{ is a field access joinpoint} \end{cases}$$

Then, given g, the set $M_I \subseteq M$ of impacted methods is defined by:

$$M_I = \{m \in M | \exists c \in C, \exists j \in f_{over}(c), g(j) = m\}$$

Figure 3 depicts how the set M_I of impacted methods is obtained from the combination of f_{over} and g. First f_{over} provides the set of joinpoints J_s from

```
1  public class TestAuction {
2        @org.junit.Test
3        public void testClose() {
4              Auction a = new Auction(..);
5              a.open();
6              a.bid(user1,30);
7              a.bid(user2,50);
8              a.close();
9              assertTrue(a.isClosed());
10             assertEquals(user2,a.getBuyer());
11       }
12 }
```

Listing 1.3. An example of JUnit test case for the `Auction` class.

the PCD `getAmount()` (A), $J_s = f_{over}(A)$. This part is performed by AJDT. Then, the set $M_I = g(J_s)$ is determined by Vidock, in which g is implemented. In this case $g(j_1) =$ `Open.open()`, $g(j_2) = g(j_3) =$ `Bid.finalize` and $g(j_4) =$ `Bid.bid()`. These methods are impacted.

3.2 Impacted test cases

The second step of the impact analysis consists in detecting the test cases that are impacted by the introduction of aspects. We want to know if a test case can reach an impacted method or not, to know if its control or data flow can be modified. First, we need to know which methods are reachable by each test case. To do so we build a static call graph for each test case.

Static call graph A static call graph is a triple (V, E_c, E_p) where:

- V is the set of vertices for methods, each method is represented by only one vertex.
- $E_c \subseteq V \times V$ is a set of edges that represent method calls. If $(m1, m2) \in E_c$, it means that the method represented by $m1$ explicitly calls the method represented by $m2$.
- $E_p \subseteq V \times V$ is a set of edges that represents potential method calls. If $(m1, m2) \in E_p$, then the method represented by $m1$ potentially calls the method represented by $m2$.
- $E_c \cap E_p = \emptyset$.

Explicit and Potential Calls A method $m1$ explicitly calls the method $m2$ if $m2$ is explicitly invoked in the source code of $m1$. The method $m1$ potentially calls $m2$ if it calls a method overridden by $m2$. We call it a *potential* call because we do not know statically which method is actually executed. The `Auction` class uses the state pattern – as illustrated by Figure 5 –, so when `Auction.close` is called, the method `close` of the current `State` object is called, but the method that is actually executed could be any of the overriding methods of `Pending`, `Open`, `Closed` and `Cancelled` (the four possible states for an auction).

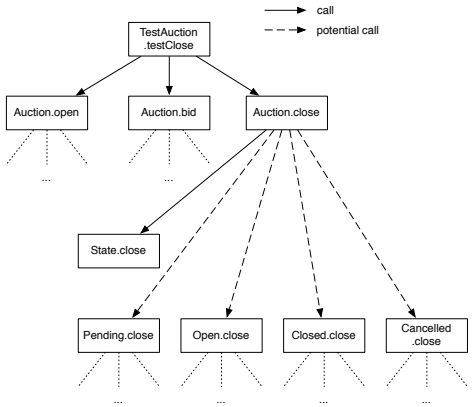

Fig. 4. The static call graph of the test case from Figure 1.3.

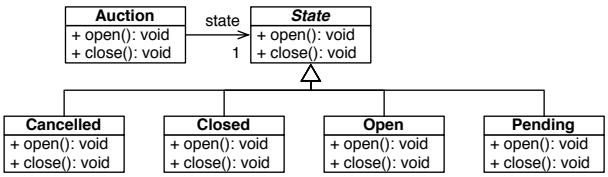

Fig. 5. Class diagram illustrating the implementation of the state design pattern.

Listing 1.3 shows an example of a JUnit test case. The test case tests the `close` method of class `Auction`. First an auction is created and two users place a bid. Then the auction is closed and the assertions check if the auction is actually closed and if the buyer is the highest bidder. Figure 4 shows an excerpt of the static call graph of this test case, with the calls and potential calls.

Vidock uses Spoon, a tool for static analysis of Java programs, to build the static call graphs. It provides a visitor pattern to explore the abstract syntax trees (AST) of the java source files. This allows the extraction of the static call graph with only the call edges (no potential calls). To obtain the potential calls we must explore the ASTs of the whole system looking for methods overriding the method targeted by a call. Potential calls are also extracted using Spoon. An inheritance tree is built to obtain the subclasses of a class. Then, for each call, we check if any of the subclasses of the targeted method's class overrides the targeted method. For each overriding methods found, a potential call is created with the source of the explicit call as source and the overriding method as target.

Impacted test case An impacted test case is a test case that can potentially execute an impacted method. Let t be a test case, S its static call graph, T_I the set of impacted test cases and M_I the set of impacted methods:

$$t \in T_I \Leftrightarrow \exists P = t, v_1, \ldots, v_n \text{ a path in } S, v_n \in M_I$$

```
1  boolean impacted(Vertex v) {
2    if onImpactedPath.contains(v) or impactedMethods.contains(v.method) then
3      return true
4    end
5    beingVisited.add(v)
6    for each v' in v.next do
7      if not beingVisited.contains(v') then
8        if impacted(v') then
9          onImpactedPath.add(v')
10         beingVisited.remove(v)
11         return true
12       end
13     end
14   end
15   visited.remove(v)
16   return false
17 }
```

Listing 1.4. The algorithm determining whether a test case is impacted or not.

A test case is impacted if, from the root of its static call graph, it is possible to reach an impacted method. Once the static call graph of a test case has been built we can determine whether it is impacted or not by using the algorithm on Listing 1.4. This algorithm returns true if the argument vertex represents an impacted method or can reach such a vertex. If we call this algorithm with the vertex representing a test case it returns true if it is impacted.

This algorithm implements a depth-first search on a graph and thus has a time complexity in $\mathcal{O}(|V| + |E_c| + |E_p|)$, where V is the set of vertices of the graph, and $E_c \cup E_p$ the set of edges, with $E_c \cap E_p = \emptyset$.

The `impactedMethods` attribute contains all the impacted methods. On line 2, the algorithm checks if the current vertex represents an impacted methods, and returns true if it is the case.

The `v.next` is the set $\{v' | (v, v') \in E_c \cup E_p\}$. On line 6, the loop calls the algorithm on the vertices that can be reached with a call or potential call edge.

A vertex that can reach a vertex representing an impacted method is said to be on an impacted path. The attribute `onImpactedPath` contains such vertices and is used to optimize the algorithm. Several test cases will most likely execute the same methods, so if we know from a previous execution of the algorithm that a vertex is on an impacted path we can stop the algorithm and return true.

Computing the complete static call graphs of all test cases can be very time consuming. This is why we build the static call graph *on the fly*. Actually the set `v.next` is only built when needed, so if we quickly find an impacted method, we do not have to find all the potential calls of the static call graph. If the test case is not impacted we cannot avoid building the complete static call graph.

The notion of potential calls may lead to an over-approximation. When there are potential calls, several of them are not executed at all at runtime, but we cannot statically remove them. So if one of this *false* potential calls can reach an impacted method, then the test case can be selected as impacted without being actually impacted. Note that this over-approximation leads to *false* positive but not to *false* negative (an impacted test case not detected as impacted).

For instance, in the example of Figure 4, the method `Open.close` is impacted by the `Reserve` aspect, and this method is actually executed so the test case is actually impacted (and detected as such). But suppose that the aspect does not impact `Open.close` but `Pending.close`, then the test case would not be actually impacted but still would be detected.

4 Validation of hypotheses

Our proposal is based on three hypotheses. First, we assume that the base program can be tested in isolation, before the aspect weaving. Thus we assume that the base program can compile without the aspects – which is only possible if the base program is not aware of the aspects. Second, the impact analysis is based on two over-approximations. (1) The set of impacted methods is over approximated in case of dynamic-PCDs, and (2) the static call graph for a test case is over approximated in presence of overridden methods. To evaluate the impact of these hypotheses on the proposed analysis, we have studied their occurrence over a set of 46 open source aspect-oriented projects. This study is detailed in a previously published paper [3].

We have selected the projects according to the following criteria: (1) Project implemented in Java/AspectJ, (2) project source code publicly available, (3) project compilable using the AspectJ compiler version 1.5, and (4) project size of at least 10 classes and 1 aspect. The 46 projects fulfilling these criteria were gathered from open source repositories. We started our search at sourceforge.net, the most popular open source repository in Internet. Out of 74 aspect-oriented projects, only 29 fulfilled our criteria. Then, we continued gathering projects by inspecting other repositories by using the Google™ code search engine. Out of 2000 files, equivalent to 31 projects, only 17 were fulfilling our selection criteria. Finally, we successfully gathered 46 open source aspect-oriented projects of size ranging from 10 to 913 classes and 1 to 64 aspects.

4.1 Hypothesis 1: Project compilation

The first hypothesis we have formulated is that aspect-oriented projects can compile without aspects. To confirm whether this hypothesis occurs in real aspect-oriented projects we have studied its occurrence over the 46 open source projects.

To know the amount of projects compiling without aspects, we have disable the aspectJ capability of each project. After re-compiling each project without aspect support, we have obtained the following results: Out of 46 aspect-oriented projects, 30 are compilable in the absence of aspects. That is, the 65% of the projects compile without the crosscutting functionality added by aspects.

The results obtained in this experience endorse our hypothesis. The percentage of projects effectively compiling without aspects is more than half of the total amount of projects. Thus, most of the projects are analyzable using our tool. Moreover, an important portion of the compilable projects range form large and medium size (between 3000 and 80900 LOC).

4.2 Hypothesis 2: Low method overriding

The second hypothesis we formulated is that the method overriding in general have a low frequency. To determine the frequency of overriding, we have studied the method overriding practice on the 46 open source projects.

We have used the Metrics [3] eclipse-plugin to analyze the java sources in each project project. The results obtained from this analysis are the following: Out of 58184 methods scattered in 8052 classes (total of 46 projects), only 3295 are overridden by subclasses. On average there are 1265 methods per projects and only 72 of them are overridden. That is, only a 6% of the total amount of methods (an average of 5,6% per project, range from 0% to 14% per project) were overridden, leaving 94% of them with no further change by subclasses.

These results widely support our hypothesis. The 6% of overridden methods is negligibly small. The over-approximation we applied in the case of inheritance overriding methods is reasonable, given the small amount of overridden methods. A common practice when building test cases is to call directly a method in a subclass instead of the method in a superclass. This allows the tester to predict the expected behavior and build a precise oracle. Thus, the impact of over approximation in the test case static call graph should be small.

4.3 Hypothesis 3: Low usage of dynamic-PCDs

The third and final hypothesis we formulated for our analysis is that the dynamic-PCDs are rarely used. To determine if dynamic-PCDs are used, we have studied the taxonomy of the PCDs declared in the aspects of the 46 open source projects.

We have analyzed each project with a custom tool [4] inspecting the PCDs definition and specifically their taxonomy. We have obtained the following results: Out of 1145 PCDs declared on 498 aspects (total of 46 projects), only 206 contained dynamic expressions (`cflow`, `cflowbelow`, `if`). That is, only a 17% of the total amount of PCDs contained dynamic expressions. Moreover, the dynamic-PCDs were present only in 15 out of 46 projects.

These results confirm our hypothesis. The 17% of dynamic-PCDs reveals that they are rarely used in real life open-source projects. Moreover their occurrence only in the 33% of the projects advocates that an over-approximation for impact analysis is reasonable. In section 6.2 we observe the effects of these hypotheses on actual impact analyses.

5 Implementation

Vidock has been implemented as a plug-in for the Eclipse IDE and is available on the internet[5]. As seen previously, the tool relies on AJDT[6] to resolve the

[3] http://metrics.sourceforge.net/
[4] `http://contract4aj.gforge.inria.fr/analysis`
[5] `http://www.irisa.fr/triskell/Softwares/protos/vidock/`
[6] `http://www.eclipse.org/ajdt/`

Example	TC	Aspect	Impacted TC	Precision	Recall
Introduction	7	Cloneable	0	NA	NA
		Comparable	0	NA	NA
		Hashable	0	NA	NA
Bean	7	BoundPoint	6 (85.7%)	100%	100%
Telecom	33	Timing	3 (9.1%)	100%	100%
		Billing	3 (9.1%)	100%	100%
Tetris	45	TestAspect	8 (17.8%)	100%	100%
		NewBlocks	7 (15.6%)	100%	100%
		Counters	2 (4.4%)	100%	100%
		Levels	2 (4.4%)	100%	100%
		All aspects	17 (37.8%)	100%	100%
Auction	306	Reserve	11 (3.6%) 7* (2.3%)	63.6%	100%
		AltBid	49 (16.0%) 39* (12.7%)	79.6%	100%
		Log	306 (100%)	100%	100%
		Reserve + AltBid	49 (16.0%) 39* (12.7%)	79.6%	100%

Table 1. Results of our experiments: added aspects and their impact on the test cases (*: actually impacted test cases).

PCDs and obtain the matched joinpoints and on Spoon [2] to obtain an abstract syntax tree of the system that allows us to build the static call graphs.

After each compilation of an AspectJ project, the plug-in applies the process described in section 3, and for each test cases that is detected as impacted a warning is reported and appears as a regular Eclipse warning. At the compilation, AJDT resolves the PCDs so we can obtain the matched joinpoints, as explained on figure 1. Then we use spoon to detect the JUnit test cases within the classes of the project. If a test case is detected as impacted we use Spoon to report a warning.

6 Case studies

To validate our approach we have experimented our analysis on several examples. Some of them are distributed with AspectJ (the *Introduction*, *Bean* and *Telecom* examples), one is a classic project freely available on the Internet (*Tetris*) and the last one has been developed by our group (*Auction*).

6.1 Description of the examples

The *Introduction* example has a unique class `Point` that encapsulate the two-dimensions coordinates of a point (using either Cartesian or polar coordinate system). Three aspects are written for that class, *Cloneable*, *Comparable* and

Hashable. All these aspects declare that `Point` implements a new interface and adds inter-type definitions to add the implementation of new methods.

The *Bean* example also implements a `Point` class. The only aspect declares that `Point` implements `Serializable` and adds bound properties. Listeners can be registered with an observer design pattern and an advice is woven around each setter of `Point` to notify the listeners.

The *Telecom* example simulates telephonic communications and handles local and long distance calls as communications with more than two interlocutors. The `timing` aspect calculates the duration connection and the customer's cumulative connection time. The `billing` aspect relies on `timing` and calculates the cost of a connection.

The *Tetris* example is an implementation of the classic video game that has been developed by Gustav Evertsson[7]. The `TestAspect` aspect traces the images that are loaded and prints their name in the console. The `NewBlocks` aspect adds new kind of blocks to the game. The `Counters` aspect counts the deleted lines and prints them on the game layout. The `Levels` aspects adds a level system to the game: each time a certain number of lines has been deleted the level is increased and the blocks fall faster.

Finally the *Auction* example is an implementation of an online auction system where users can buy or sell items. The implementation uses the command and the state design pattern [4]. The command pattern is used by the server to process the instruction received by the clients. The state pattern is used to represent the state of an auction. There are 4 different states (pending, open, closed and cancelled). The system (without the test cases) has 1382 lines of code, 41 classes. The aspects that have been added are those presented in section 2.1.

6.2 Results

For each example, test cases have been written using statement coverage as a minimum criterion. These test cases allowed us to detect and fix several errors. For the auction system we validated the program. In the *Introduction* example of AspectJ we detected several bugs with the handling of polar coordinates.

Then the aspects have been added to the base program, alone if possible, with the depending aspects otherwise. After each weaving the impact analysis has been performed on the test cases. The results are summarized in Table 1 and are discussed below. In the following, we first discuss the extreme results – 0% or 100% of impacted test cases –, then we discuss the results of most cases.

For the *Introduction* example none of the test cases are impacted by the aspects. This result happens because the aspects do not add any advice so none of the test cases are impacted. Also, the methods introduced by the aspects are not executed by the test cases for the base class.

The `log` aspect of *Auction* impacts all the test cases but this is a particular case: the aspect is woven everywhere in the code, before each method execution.

[7] `http://www.guzzzt.com/coding/aspecttetris.shtml`

When the system has few classes, the aspects are more likely to impact a large part of the test cases. In the *Bean* example the `BoundPoint` impacts 6 out of 7 test cases. This aspect defines only one advice which is woven around methods that are called by many other methods of `Point`. This particular case, where a few impacted methods are called by almost every method in the system will more likely happen in a small system.

Apart from these borderline cases, less than 20% of the test cases are impacted, and in some cases less than 5%. These results show that, in our examples, our analysis can save execution time and help the evolution of the test cases. Execution time is saved by avoiding the execution of the non-impacted test cases. This analysis helps the evolution because the programmer can focus on the impacted test cases to determine whether they should be modified or not as the non-impacted test cases can be ignored.

To estimate the effects of the over-approximation, we have computed the precision and recall for each aspect. Each test case was manually checked to see if it was impacted. The recall is always 100%, which means that are no *false negative* (i.e., impacted test cases not selected). In most cases the precision is 100%, which means that there are no *false positive* (i.e., selected test cases that are not impacted). With the `Reserve` aspect, the precision is only 63.6%, but there are only four false positives. In the other two cases where there are false positives, the precision is good (79.6%). These results show that the over-approximation has little effect and no impacted test cases are missed.

When several aspects are added simultaneously, the number of impacted cases is not always the sum of the test cases impacted independently by each aspect. Table 1 shows the results when all the aspects of *Tetris* are added and when `Reserve` and `AltBid` are added in *Auction* – note that in *Telecom*, `Billing` relies on `Timing`, so when `Billing` is added both are added.

This shows that the number of impacted test cases does not necessarily grow with the number of aspects in the system. In *Tetris*, the number of test cases impacted by all the test cases (17) is a bit less than the sum of the test cases impacted by each aspect (19). In *Auction*, all the test cases impacted by `Reserve` are also impacted by `AltBid`, so the number of test cases impacted by the two aspects is equal to the number of test cases impacted by `AltBid`.

7 Related Work

Several work have proposed static analysis to evaluate the impact of changes in a program. Vokolos *et al.* [5] have introduced a tool for regression test selection on C programs. It selects test cases based on a textual comparison of the source files. Rothermel *et al.* [6] have presented an algorithm for regression testing that compares control flow graphs (CFGs) and thus is less dependent on the programming language. It requires a CFG for each version of the program and coverage informations for each test case. If a test case covers a part of the initial CFG that differs in the new CFG then the test case is selected. This algorithm has been adapted to Java programs by Harrold *et al.* [7], and then to AspectJ by

Xu *et al.* [8]. Zhang *et al.* [9] have developed a change impact analysis tool for AspectJ, Celadon. It detects atomic changes in the source code of the program by comparing the ASTs of the different versions. Then call graphs of the test cases are produced to detect the test cases affected by the change.

These static analysis are close to the analysis of Vidock but they differ because they evaluate the impact of a change whereas Vidock focuses the impact of the introduction of aspects. Vidock relies on the PCD to locate the impacted methods, whereas other techniques compare two models of the program (e.g., ASTs, CFGs) to detect which parts have changed. This results in complex graph comparison algorithms: according to Rothermel *et al.* [6], their algorithm is quadratic in the worst case. Also, the algorithm of Rothermel *et al.* requires the coverage information of the test cases on the previous version of the program. Vidock is more focused, but in the specific case of aspect weaving, proposes an efficient solution for change localization.

Ren *et al.* [10] have proposed a change impact analysis tool for Java programs. This work is different because the goal is to compute the subset of changes that affect a test cases, instead of computing the subset of test cases that are affected by all the changes. This approach uses a dynamic analysis, to detect at runtime the changes that are executed by a test case. The changes are detected statically.

8 Conclusion

In this paper we have presented Vidock, a tool for the impact analysis of aspect weaving on test cases. It performs a static analysis that identifies the subset of test cases that are impacted by the aspect weaving.

To ensure that we do not miss impacted test cases, the analysis over-approximates the set of impacted test cases in the case of dynamic poincut descriptors and polymorphism. Thus we can have *false positive* (test cases detected as impacted that are actually not impacted) but no *false negative* test cases (impacted test cases that are not detected).

To validate the ability of Vidock to detect impacted test cases and the effect of over approximation on the Vidock's results, we have performed two types of studies. First, we have experimented Vidock on 5 systems. The experiments showed that in most cases only a few test cases are impacted by the aspects. They also showed that the over-approximations have a minimal effect on the results. Second, we have studied 46 open-source AspectJ projects to evaluate the occurences of dynamic PCDs and method overriding. We observed that 65% of the projects can compile without aspects, and thus the *base program* can be tested in isolation; 6% of the methods are overridden and 17% of the PCDs are dynamic, so the effect of over-approximations should be minimal in general.

Vidock has been developed as a plug-in for the Eclipse IDE. It performs the impact analysis automatically after each compilation and reports a warning for each impacted test cases. This tool is available for download.

In future work we want to investigate aspects classifications such as the one proposed by Rinard *et al.* [11] or Munoz *et al.* [12], to assist the evolution of

impacted test cases. Munoz *et al.* propose a classification based on the impact of an aspect on the control flow or data flow. Using this information, we can know, for example, that test cases impacted by an aspect that does not modify the control and data flow must be executed and must pass. If these test cases fail, we can locate the fault in the aspect that has been introduced. Also, better understanding the impact of an aspect will probably help the evolution of the test cases. If an aspect is independent of the base code – i.e. neither the aspect nor the base program may write a field that the other may read or write – the oracle of the impacted test cases most likely will need to be augmented but the existing assertions should remain unchanged.

References

1. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison Wesley (2005)
2. Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA (May 2006) http://spoon.gforge.inria.fr.
3. Munoz, F., Baudry, B., Delamare, R., Le Traon, Y.: Inquiring the usage of aspect-oriented programming: an empirical study. In: ICSM '09: Proceedings of the 25th International Conference on Software Maintenance. (2009)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
5. Vokolos, F.I., Frankl, P.G.: Pythia: a regression test selection tool based on textual differencing. In: 3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems, London, UK, UK, Chapman & Hall, Ltd. (1997) 3–21
6. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. **6**(2) (1997) 173–210
7. Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A.: Regression test selection for java software. In: OOPSLA'01: Proceedings of the 16th conference on Object-Oriented Programming, Systems, Languages, and Applications. (2001) 312–326
8. Xu, G., Rountev, A.: Regression test selection for AspectJ software. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering. (2007) 65–74
9. Zhang, S., Gu, Z., Lin, Y., Zhao, J.: Celadon: a change impact analysis tool for aspect-oriented programs. In: ICSE Companion '08: Companion of the 30th international conference on Software engineering. (2008)
10. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of Java programs. In: OOPSLA '04. (2004) 432–448
11. Rinard, M., Salcianu, A., Bugrara, S.: A classification system and analysis for aspect-oriented programs. SIGSOFT Softw. Eng. Notes **29**(6) (2004) 147–158
12. Munoz, F., Baudry, B., Barais, O.: Improving maintenance in aop through an interaction specification framework. In: ICSM08, 24th International conference on Software Maintentance, Beijing, China, IEEE Computer Society Press (2008)