Reasoning about Faults in Aspect-Oriented Programs: A Metrics-based Evaluation

Rachel Burrows*, François Taïani*, Alessandro Garcia† and Fabiano Cutigi Ferrari‡
*School of Computing and Communications – Lancaster University – UK
Email: {r.burrows, francois.taiani}@comp.lancs.ac.uk
†Informatics Department – Pontifical Catholic University of Rio de Janeiro – Brazil
Email: afgarcia@inf.puc-rio.br
‡Computing Department – Federal University of São Carlos – Brazil
Email: fabiano@dc.ufscar.br

***Abstract*—Aspect-oriented programming (AOP) aims at facilitating program comprehension and maintenance in the presence of crosscutting concerns. Aspect code is often introduced and extended as the software projects evolve. Unfortunately, we still lack a good understanding of how faults are introduced in evolving aspect-oriented programs. More importantly, there is little knowledge whether existing metrics are related to typical fault introduction processes in evolving aspect-oriented code. This paper presents an exploratory study focused on the analysis of how faults are introduced during maintenance tasks involving aspects. The results indicate a recurring set of fault patterns in this context, which can better inform the design of future metrics for AOP. We also pinpoint AOP-specific fault categories which are difficult to detect with popular metrics for fault-proneness, such as coupling and code churn.**

***Keywords*-Aspect-Oriented Programming; Fault-proneness; Software Metrics**

I. Introduction

Aspect-Oriented Programming (AOP) [1] seeks to facilitate program comprehension by providing mechanisms that improve modularity of crosscutting concerns. Because aspects are usually included and extended as the software projects evolves, the reliability of aspect-oriented (AO) code needs to be better understood in this context. In particular, the characteristics of the source code of such evolving projects need to be measured to better highlight their impact on program understanding and fault-proneness. Traditionally, such measurements rely on code metrics, which are then used to detect potentially faulty modules. However, even with the establishment of industry-strength AOP frameworks such as SpringAOP, JBossAOP and Glassbox, the metrics-based detection of fault-prone modules in evolving aspect-oriented software has rarely been investigated.

Contemporary evidence suggests that faults are largely influenced by particularities of the underlying programming mechanisms [2, 3, 4, 5]. The potential downside of AOP mechanisms is that they introduce intricate dependencies between modules, which might in turn lead to faults [6, 7, 8, 9, 10]. The role of these dependencies in the fault introduction process is particularly important to during software maintenance tasks, when developers must first understand the dependencies between modules of programs. It is also important in developing effective testing strategies for AOP [11, 12, 13]. Evaluating and improving existing metrics, and the general AOP practice, requires however a good understanding of how faults might arise when AO programs are maintained. Unfortunately, such an understanding is still missing for AOP, in particular during maintenance tasks involving aspects.

This paper presents the results of an exploratory user study focusing on the faults introduced by maintenance tasks in AO programs. We asked 16 developers, organised in eight pairs, to conduct specific maintenance tasks on an existing AO application. The application is written in AspectJ, a language that provides a classical AOP model [14] that includes, mainly, pointcut and advice mechanisms. A tally of approximately 130 changes made by programmers were analysed and tested to reveal faults. We report how those faults tended to be introduced by programmers while including or extending aspect code in the target program. In addition, we systematically analyse to what extent some classical metrics [15, 16, 17] for fault-proneness could be used to detect the faulty aspects. Based on these analyses, our main findings are as follows:

1) By far, faults related to incorrect join point selection occurred when introducing new pointcuts rather than when partially or fully reusing a pointcut.
2) More importantly, we also found that AOP-specific coupling metrics proposed in earlier works [16, 10] do not correlate well with pointcut faults in evolving programs. Surprisingly, these faults do not correlate to code churn either, a traditional metric that is typically highly correlated with faults [17, 18];
3) Many faults were introduced when data dependencies between base classes and aspects increased. In particular, directly extracting data from a base code module appeared to be more fault-prone than other methods of data extraction e.g. using shared join points or reflection.
4) We use this analysis to highlight future directions for AOP metrics. In particular, we show that a specific

targeting of the mechanism being measured, and *lower granularity* of measurement can substantially improve the fault sensitivity of a metric, and propose a new metric to detect pointcut faults based on these principles.

The paper is structured as follows. After a brief background on AOP, fault proneness, and metrics (Section II), we present our experimental setup (Section III). We then analyse the observed fault patterns, and propose a new metrics to address the limitations we noticed in popular metrics (Section IV). In the sequence, we summarise the related research and discuss the limitations of our work (Section V). Finally, we reflect on future directions for AOP metrics in our conclusion (Section VI).

II. Background and Problem Statement

A. Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [1] seeks to improve the modularity of crosscutting concerns, such as exception handling, concurrency and caching. Whereas these concerns are typically scattered across multiple modules in a software system, AOP offers mechanisms to refactor this code from the *base code* and modularise them into *aspects*.

An aspect contains *advices*, which are method-like constructs that encapsulate crosscutting behaviour. Differently from methods, advices are implicitly invoked[1] at specific points of the program execution (the *join points*), which are specified in *pointcut expressions*. AOP languages such as AspectJ also support intertype declarations (ITDs) that allow aspects to alter the static structure of the system, e.g. by introducing new members into modules such as methods or fields and modifying the inheritance amongst classes through declare-like statements. More details about the language features can be found in the language documentation [19].

B. Fault Prediction in AO Programs

AOP may improve important software quality attributes such as evolvability [20], modularity [21] and maintainability [22]. However, it also introduces intricate dependencies between aspects and the base code. This can lead to the introduction of specific faults which are particularly hard to foresee or detect with conventional coupling metrics [23, 24, 25]. For instance, a pointcut expression has the powerful ability to quantify and implicitly invoke an advice method at multiple points, significantly changing the control flow and the data flow of the underlying program. This greatly hardens the task of predicting, detecting and locating faults in AO applications.

We highlight that, in general, faults are largely influenced by particularities of inter-module dependencies established by the underlying programming mechanisms [2, 3, 4, 5]. Despite a large body of work evaluating module dependencies and faulty modules in object-oriented programs, this level of knowledge is lacking with respect to AOP, for which only a few initiatives can be found [10, 9].

C. Coupling and Churn Metrics for AO Programs

Coupling metrics are popular indicators of fault-proneness in object-oriented systems [2, 3, 4]; Existing work has highlighted effective coupling metrics as indicators of good modularity such as Coupling Between Components and Depth of Inheritance Tree [15].

In the context of AOP, one can find a number of coupling metrics [16, 26, 23, 27] which have been successfully used in empirical studies of AOP [26, 28, 29]. However, they have certain limitations for not effectively capturing subtle coupling unique to AO programs that results from the use of the aforementioned constructs [23, 24, 25].

In our previous research, we found that certain coupling connections are more fault-prone than others [9]. We then proposed a suite of exploratory metrics [10] that showed to be more effective for predicting faulty modules than traditional (AOP-adapted) metrics such as Coupling Between Components and Coupling on Advice Execution [16]. Obviously, the achieved findings require further investigation in order to allow proper generalisation.

In regard to code churn metrics, they have recently shown high capacity of fault prediction [17, 18]. In the AOP field, code churn has been investigated as a predictor of design flaws in studies about design stability [28, 29]. Despite the work by Stoerzer and Graf [30], who explore delta analysis to realise the impact of changes in pointcut expression in terms of quantification of join points, to the best of our knowledge code churn has not been investigated as fault predictor for AO programs.

III. Study Setup

A. Study Goal and Hypothesis

The goal of this study is to explore fault introduction processes in aspect-oriented programs and compare the effectiveness of existing coupling and churn metrics for AOP to detect the faulty modules. As part of the analysis we aim to analyse the underlying causes of common faults in AO programs and compare the ability of existing metrics to detect fault-prone implementation strategies in such programs.

B. The Target Application

In this study, we evaluated an AspectJ application called Telecom, which is a telephony system simulator which is originally distributed with AspectJ [19]. We planned a set of maintenance tasks in Telecom– further described in Section III-C – with the goal of understanding how faults are introduced while the system evolves.

We carefully selected the Telecom application as it provides suitable base template to investigate a variety of dependencies between both the classes and aspects. The aspects contain advice, pointcuts, declarations which change the control and data flow of the base program in a variety

[1] This is the implementation model found in AspectJ-like languages, although this may vary in other AOP supporting technologies.

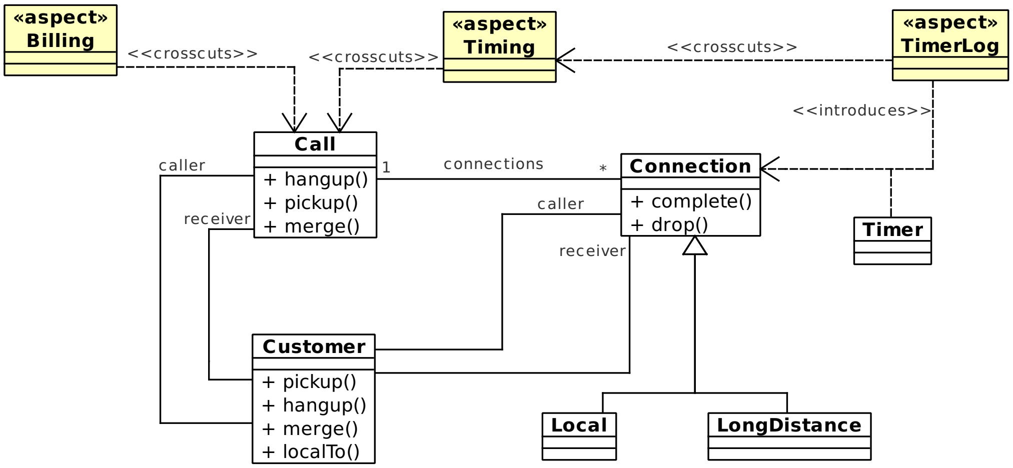


Figure 1. UML representation of classes and aspects in the Telecom system.

of ways. Importantly, the example is not too large or overly-complex, thus enabling us to reduce the effect of extraneous variables. It also enables us to perform a deep analysis of the faults related to maintenance tasks as described further in section III-C.

More importantly, the chosen application can be used in the context of controlled experiments where time is constrained for the programmers to comprehend and perform the tasks they are assigned.

Figure 1 shows a simplified UML diagram of the system. In Telecom, timing and billing of phone calls are handled by aspects. The version we use in this study includes nine modules, being six classes and three aspects.

The `Call`, `Connection`, and `Customer` classes provide the basic functionality, i.e. they simulate local and long distance phone calls between customers. The `Timing` aspect measures the duration of calls, which is logged by the `TimerLog` aspect. `Billing` implements the billing concern and ensures calls are charged accordingly, managing updates in the customers' bill.

Note that Figure 1 shows a simplified view of the system, which includes part of the public interface of the classes and the modules that are crosscut by aspects[2].

C. Setup and Coding Tasks

The study included 16 developers with experience in Java, AOP and AspectJ. The level of experience varied, some developers only possessed academic-level experience with both programming languages whereas some participants had industrial experience.

Pair programming was utilised to promote discussion on alternative designs and AO implementing strategies (e.g. which join point should be selected and whether or not a pointcut should be reused) . Each pair of developers was given a set of 11 maintenance tasks to perform on the Telecom system. Each developer was previously given a briefing on the functionality of the Telecom application and a presentation stating the main role of each module.

The maintenance tasks – listed in Table I – did not specify the coding strategy to be followed by the developers. Therefore, they intended and resulted in developers modifying different modules and varying the types of dependencies sourced from a broad range of AOP mechanisms. The time allocated to perform the tasks was 90 minutes, although developers were able to submit their answers if they finished beforehand.

Note that, in order to aid understandability, the tasks are divided into three groups, namely T1, T2 and T3. Each of them contains one or more maintenance operation, which are enumerated throughout the text included in the table. For instance, #1 marks maintenance operation 1, #2 marks maintenance operation 2 and so on.

The defined tasks included modifying existing functionality (T1 and T2), and adding new functionality (T3, #4–#11). Maintenance operarations from T1 did not require any access to data from the base code whereas operations from T3 (#5–#10) did; the functionality of T3 could be achieved by matching a number of different join points, creating new pointcuts or reusing existing ones, and using ITDs and different types of advices. This allowed an in-depth comparison of the fault-proneness of a variety of implementation strategies.

The focus of this study is both qualitative and quantitative in nature. We performed an direct comparison of the different implementations in order to find trends in the types of faults introduced. To aid the qualitative analysis, we performed Pearson's correlation analysis. This enabled us to quantitatively assess the effectiveness of metrics in indicating faulty modules.

D. Fault Collection

Existing JUnit tests were utilised from the work of Lemos et al. [31]. These were adapted and extended in order to test

[2] Note that Figure 1 does not show advices within the aspect elements since advices in AspectJ are anonymous method-like constructions.

Table I
ENUMERATED DESCRIPTION OF MAINTENANCE TASKS

Task	Description
T1	Manipulate the `TimerLog` aspect so that when any method in the `Timer` is called, a message is printed out. The message to be printed out is “Timer class is being accessed”$_{\#1}$
T2	Manipulate the `Billing` aspect so that both the caller$_{\#2}$ and receiver$_{\#3}$ share the cost of the call.
T3	Record call information using a new aspect named “`CallHistory`”.$_{\#4}$ The information to be saved per call is as follows: – Name of the caller$_{\#5}$ and receiver$_{\#6}$. – Duration of the call between caller$_{\#7}$ and receiver$_{\#8}$; to be saved in a variable named “`totalConnectTime`”. – Charge given to the caller$_{\#9}$ and receiver$_{\#10}$ for the call. Test your `CallHistory` aspect works correctly; create a `callsToString` method, which prints out all saved information to the screen.$_{\#11}$

the new functionalities introduced by the tasks. These tests aimed to validate and verify (i) functional core requirements of the application e.g. making a call between customers; (ii) behaviour of the existing aspects e.g. billing and timing semantics; and (iii) behaviour added from maintenance tasks completed by the developers.

Figure 2 presents the structural coverage[3] achieved with the developed test set. Note that all implementations, enumerated as G1–G8, have been executed on the same test set, irrespective of their differences. The chart shows that the statement coverage is higher than 90% on average, while the branch coverage is between 70% and 80% on average.

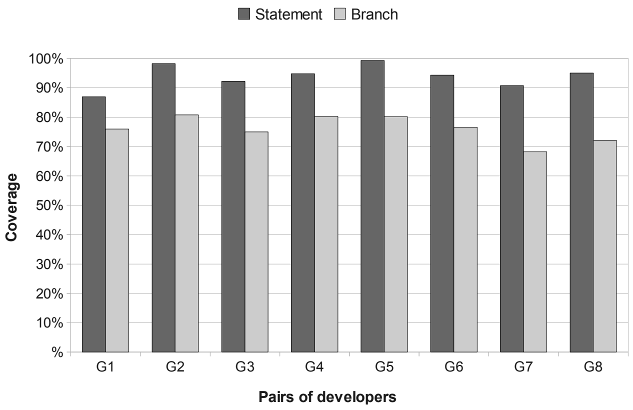


Figure 2. Structural test coverage of the implementations.

A total of 26 faults were reported and categorised according to a revised fault taxonomy for AO software [8]. This taxonomy classifies each fault according to its cause. For instance, a fault can be related to (i) a pointcut expression; (ii) an ITD or a declare-like statement; (iii) an advice; or (iv) the base program. The fault distribution per category is depicted in Figure 3. Note that the two base program-related faults occurred due to changes in the base modules performed by the developers during the maintenance tasks, that is, they were not present in code provided to the developers.

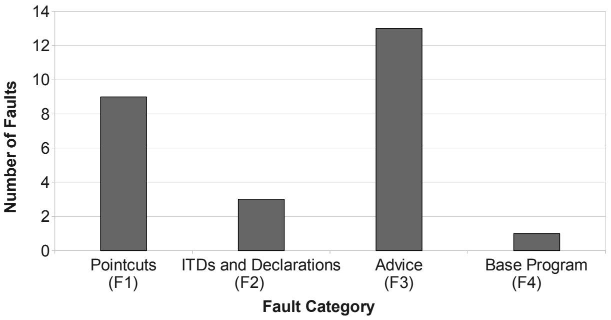


Figure 3. Number of faults per fault category.

E. The Metrics

We applied a variety of metrics in this study, which include code churn and coupling metrics. The criteria for metrics selection are based on their popularity as well as their effectiveness in indicating fault-proneness in industry-strength systems [17, 18], object-oriented [2, 3, 4, 5] programs and aspect-oriented programs [10, 9].

In our study, code churn metrics brought an important insight to the analysis as each modification has a certain likelihood of introducing a fault into the code. The code churn metrics were collected by creating a Ruby script[4] that parsed through the AspectJ source code. By comparing the original code with the modified code we quantified the total number of lines (i)added, (ii)removed, and,(iii) added or removed.

We also collected three popular coupling metrics for AO programs: *Coupling Between Components* (CBM), *Coupling on Advice Execution* (CAE) and *Depth of Inheritance Tree* (DIT) [16]. These metrics are the most commonly applied metrics for empirical studies in AOP [25] and consist in direct extensions from the popular Chidamber and Kemerer metrics suite for object-oriented programs [15].

Finally, two coupling metrics named *Base Coupling* and *Aspect Coupling* were developed in our previous work [10]. These two metrics consider different coupling directions: the coupling *induced* by an aspect on the rest of the code, yielding *Aspect Coupling*; and the coupling received by a

[3] Computed using the JaBUTi tool [31].

[4] *http://www.ruby-lang.org/* - 04/02/2011

module (class or aspect) through this mechanism from all aspects in the system, yielding *Base Coupling*. An advantage of these metrics is that it ensures we do not overlook fault-proneness sourced from either side of the aspect-base code relationship. Another key difference of these metrics is that they are more sensitive to coupling frequency. By coupling frequency we mean the number of times a module is advised (possibly at the same join point) rather than summing up the number of distinct modules that are coupled. For instance, an aspect that advises a class once is probably less fault-prone than an aspect that advises a class several times. By capturing finer-grained coupling, these metrics have shown to be more effective at indicating fault-prone modules than traditional coupling metrics [16, 26, 23, 27].

Table II summarises the metrics applied to each group's implementation, in addition to the overall number of faults we found. For each metric we show the average value across all groups with the standard deviation. Modules in which no change occurred are shown in grey.

IV. Analysis and Discussion

Existing software metrics are often not designed to capture particular implementation strategies or be more sensitive to particular types of faults. Yes, different implementation strategies might be more or less fault-prone, and might lead to different types of faults. If we better understand how software metrics can reflect the strategies and types of faults occurring in AO programs, we will be in a better position to improve both the metrics, and by feedback, proposed enhanced AO designs and mechanisms.

In the following, we first report on the different implementation strategies we observed in our study and the types of faults they led to. We then look at how well existing software metrics were able to detect these different strategies, and to provide contextual guidance on the type of potential faults in AO modules. We finally propose a new metrics that improve on the deficiencies we observed, and show that, in our study at least, it considerably enriches the set of tools available to program managers to detect fault-prone situations in AO software.

A. Strategies and faults

1) Reuse vs. Pointcut Creation: We first noticed that, when adding a new advice, developers were much more likely to capture an incorrect set of join points (categorised as an F1 fault in Figure 3) if they added a new pointcut, rather than if they reused an existing pointcut of the telecom application. In total, 9 F1 faults correspond to incorrectly selected joinpoints, and all occur in newly created pointcuts.

This is particularly visible in the `CallHistory` aspect, in which 8 of the above faults can be found (Table III). Developers were required to create a new aspect named `CallHistory` that implemented features described in tasks 4-11. In Table III, F1 faults (related to pointcuts) are categorised according to the implementation strategy used (new vs. existing pointcut). Of the 8 F1 faults, 5 of are due to a selection of a subset of intended join points, one to the selection of a wrong set of join points, which includes both intended and unintended items, and, 2 due to selection of a wrong set of join points, which included only unintended items. All 8 of these faults were found in aspects where advice was bound to new pointcuts. By comparison, when new advice added was bound to existing pointcut expressions, the pointcut expression was advising join points suitable for the advice(0 F1 faults).

Two main reasons might explain why developers were more likely to select a wrong set of joinpoints when creating a new pointcut than when binding to existing pointcuts. Firstly, if certain join points are already captured by existing pointcuts, they probably represent a crucial point in the programs execution. For instance, the `Timing` aspect has pointcuts that capture key stages of the application's execution (e.g. the start of a call and the end of a call). These key stages act somewhat as "semantic checkpoints" where essential information about the application's logic is made available. At these execution points, multiple tasks need to be completed such as calculating the cost of the call, stopping the timer, logging information. For this reason, when completing a maintenance task related to one of these key stages (such as Tasks 5 to 10), a suitable pointcut is likely to be already available in the existing code.

Secondly, if there is a suitable pointcut, its capabilities for extracting data have already been demonstrated within the existing advice thus aiding understandability for other aspects that wish to advise the same method or access the same data. Reusing pointcuts also provides developers with efficient means to control advice ordering by bringing potentially interfering advices on the same set of joinpoints, where constructs such as `declare precedence`, or the different types of advice available in AspectJ can be applied (i.e. one advice to execute before the join point and the other after). By contrast, creating a new pointcut often requires an in-depth understanding of the existing interactions between the base and aspect code. For instance, Tasks 5 to 10 required developers to extract data regarding the call that had just ended. One potential candidate to obtain this data is the method `hangup` (Figure 4). Unfortunately, when `hangup` finishes, the data about the call is not available yet: this data is produced by the aspects `Billing` and `Timing`, which execute later by advising another method that follows `hangup`. This complex set of dependencies eliminates `hangup` (and any previously executing method) as a potential join point for Tasks 5-10, but the incompatibility is hard to detect and requires an in-depth understanding of how `Billing` and `Timing` are woven into the base program.

The complexity of deciding *when* to extract data may also explain the 2 faults created to incorrect advice type specification as both pointcut expression related faults and incorrect advice type specification faults have the difficulty of deciding the point in execution in which the advice is

Table II
Overview of Results

	Module	Code Churn		Lines of Code		Coupling Between Modules		Coupling on Advice Execution		Depth of Inheritance Tree		Aspect Coupling		Base Coupling		# Faults	
		Avg	σ	Avg	σ	Avg	σ	Avg	σ	Avg	σ	Avg	σ	Avg	σ	Avg	σ
	CallHistory.aj	35.25	18.22	30.50	16.74	3.50	2.07	0.50	0.53	0.00	0.00	3.50	4.63	1.25	1.49	2.69	2.15
	Billing.aj	13.63	3.42	35.00	5.81	3.63	1.06	1.50	1.07	0.00	0.00	9.50	3.85	1.50	0.76	0.38	0.52
	TimerLog.aj	4.25	1.28	12.00	1.31	1.00	0.53	0.13	0.35	0.00	0.00	11.63	5.21	0.00	0.00	0.13	0.35
	Call.java	3.38	9.55	51.75	7.78	3.75	0.71	3.13	0.83	0.00	0.00	0.00	0.00	7.00	3.70	0.00	0.00
	Customer.java	1.13	1.55	42.75	1.39	1.25	0.71	2.50	0.76	0.00	0.00	0.00	0.00	3.88	0.64	0.00	0.00
	Timing.aj	0.38	0.74	19.00	0.00	2.75	0.71	1.25	0.71	0.00	0.00	5.38	4.31	6.38	1.69	0.06	0.18
No change	Connection.java	0.00	0.00	33.00	0.00	0.33	0.82	1.67	0.82	0.00	0.00	0.17	0.41	3.00	1.10	0.00	0.00
	Local.java	0.00	0.00	6.00	0.00	0.13	0.35	0.88	0.35	1.00	0.00	0.00	0.00	1.25	0.71	0.00	0.00
	LongDistance.java	0.00	0.00	6.00	0.00	0.13	0.35	0.88	0.35	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
	Timer.java	0.00	0.00	112.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.00	0.74	0.00	0.00

inserted into the base code.

```
// Call.java
public void hangup(Customer c) {
 for (Enumeration e =
      connections.elements();
      e.hasMoreElements();) {
   ((Connection)e.nextElement()).drop();
 }
--------------------------------
// CallHistory.aj
pointcut hangUpMethodCall(Call c) :
 call (void Call.hangup(Customer)) &&
 target(c);

before(Call c): hangUpMethodCall(c) {
 history.add(c);
}
```

Figure 4. Code Extract: Incorrect Implementation.

Table III
Faults per pointcut strategy in CallHistory aspect

Advice Binding Strategy	# of matching advices	F1 *(avg)*
new	7	7 *(1.16)*
existing	3	0 *(0.00)*

2) *Data Flow Faults:* A second pattern we noticed, was the prevalence in our study of faults that involved the extraction of data from the base program. This type of data-flow faults actually crosscuts the categories used in Figure 3, as faults may occur at any point of the data-flow path from the base-program to the aspect and (for invasive aspects) back. The wrong data might get extracted (F1 faults), the wrong attributes might be added to base modules (F2 faults), the data might be improperly processed (F3 and F4 faults).

Table IV shows the distribution of faults within the modified aspects only. The aspects are divided into 4 rows depending on their strategy for data extraction. These strategies were to either to (i)extract required data directly from the base code e.g. creating a new pointcut and advice within the aspect, (ii) indirectly via another aspect e.g. using shared join point or reflection mechanisms such as `aspectOf()`, (iii) a mix of (i) and (ii), and, finally (iv) no data extraction - for aspects that did not require any contextual information from the base code to be passed into the advice body.

Table IV
Faults per Data Access Strategy

Access Strategy	# Aspects	#Advices	#Reflection Uses	Faults
Direct Only	5	9	3	12
Direct and indirect	5	10	4	7
Indirect only	7	7	9	5.5
No Data Extraction	7	8	0	1

The key difficulty in extracting data is twofold: (i) developers need to reason about the availability and current state of the data when creating the pointcut; and (ii) developers need to reason about how to access and use the data. Therefore, if an aspect undergoes modifications, the fault-proneness is drastically increased if it involves a dependency on data from the base code.

Firstly,the pointcut signature defines any extracted data within the parameters. Secondly, the advice signature has to define any data that is to be used within the advice body. Finally, the data has to be handled properly in the advice's implementation.

Indirectly accessing data proved to be less fault-prone than the directly accessing the base code. In fact, the module with the highest number of faults was `CallHistory_CC` (see Figure 5) which had a total of 6 faults. This module was composed of three pointcuts and data was extracted from the base code in each of them.

B. *Existing Metrics*

Table V shows the Pearson Correlation Coefficient we obtained between the metrics selected for our study (Section III-E) and the faults we obtained. Of course, to generalise the results would require confirmation based on much larger datasets; however, the test indicates some interesting trends, which we discuss below. The correlation coefficients for F2 and F4 faults (related to Intertype Declarations and base code) are shown for information. However because these faults are barely represented in our study, drawing conclusions appears difficult, and in the following, we focus on F1 and F3 faults (related to pointcut and advice implementation, respectively).

Table V
PEARSON CORRELATIONS BY FAULT CATEGORIES

Metrics	#Faults	F1	F2	F3	F4
CBM	0.25	-0.09	0.29	0.33	0.26
CAE	-0.18	-0.16	0.04	-0.13	0.02
DIT	-0.19	-0.11	-0.12	-0.14	-0.07
Aspect Coupling	-0.20	-0.17	0.23	-0.20	0.04
Base Coupling	-0.09	-0.14	0.00	0.00	-0.03
Churn	**0.55**	0.19	-0.01	**0.62**	0.14

The first observation is that churn is, by far, the metrics that is best correlated metrics with the overall number of faults. Coupling between Module (CBM) comes as second, which possibly hints the role played by coupling issues in the fault introduction process. However this role is not reflected in any of the other coupling metrics we looked at, namely Coupling on Advice Execution (CAE) [16], Aspect Coupling [10], and Base Coupling [10].

The good correlation between churn and the overall number of faults confirms what previous studies have shown for industry-strength systems [17, 18]. Churn metrics are an effective way of evaluating code stability, and to indicate of fault-proneness of both in Object Oriented and AO programs. The role of churn in our study is further illustrated in Figure 5, showing the number of faults (y axis) per module produced by each group (x axis), with the modules ordered in decreasing order of churn—the majority of faults appeared in modules with high churn.

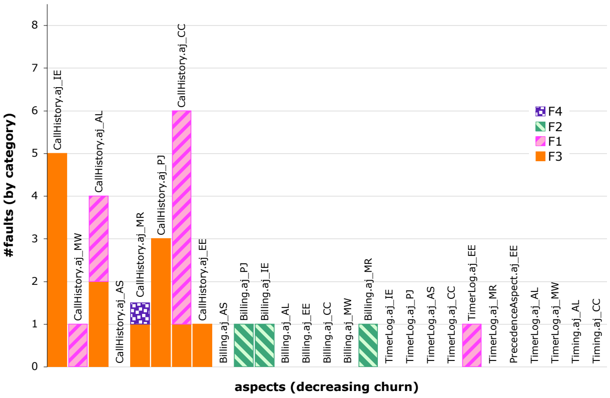


Figure 5. Churn vs Fault per Category

This good correlation of churn is however highly conditioned on the types of faults: Although it is particularly high for F3 faults (related to advise implementation, coefficient of 0.63), the correlation coefficient drops to 0.19 for F1 faults (pointcut related). This can be explained by the specific role of pointcuts in AO programs: because a pointcut expression determines the set of joinpoints to which an advise applies, comparatively small changes in a pointcut expression (in term of churn) can have wide ranging implications throughout an AO program.

Pointcuts are thus *dense* constructs. Their dense nature causes challenges both for developers (illustrated by the difficulty of introducing new pointcuts in an existing programs), and for the churn metrics. Whether small or wide ranging, a modification within a pointcut will only register as a one-line modification, barely showing up in the churn numbers. Another difficulty of many coupling and churn metrics for AOP when used as indicators of fault-proneness is that they do not take into account aspects that are data dependent. Aspects that are invasive (e.g. if they rely on extracted data from the base code) are more likely to impose a risks into the base code such as introducing a fault [32, 33, 34].

More generally, these findings demonstrate the importance AO-specific fault categories to analyse how faults are introduced in AO software. Although the general number of faults seem to follow the traditional correlation to churn (Pearson's coefficient of 0.55), this hides a chequered reality, where a substantial category of faults (F1 faults, 34%) are not accounted for, and would not be properly targeted by a churn led analysis.

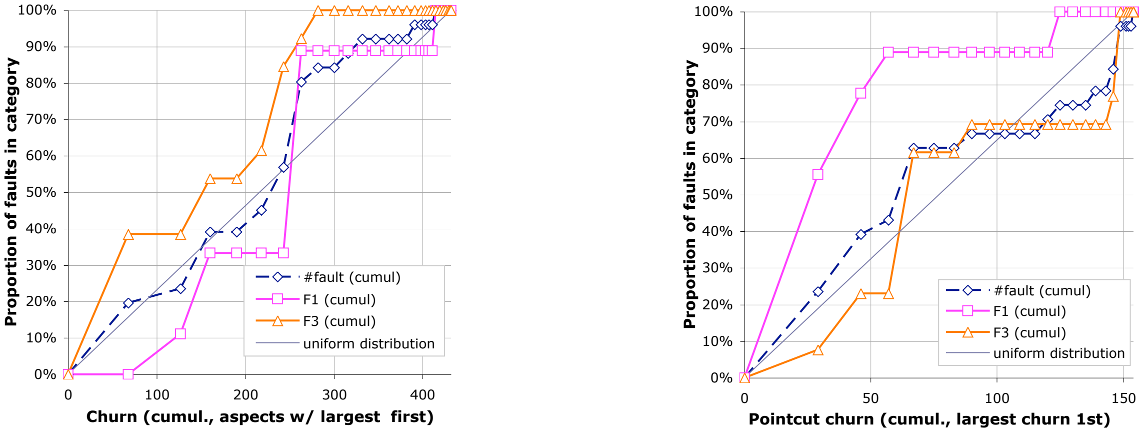
Interestingly, F1 faults are not well correlated by any of the other metrics we used, and in particular not by coupling metrics, although these were found to correlate well to faults induced by aspectisation (the transformation of a non-AO program into an AO version) in a previous study [10]. Besides the small scale of the study at hand, this can be explained by the different nature of the tasks, as here developers had to both realise the integration of their aspects in the original code (aspectisation) and realise the functionalities of these aspects. The telecom application our study uses is also characterised by narrowly-scoped (domain-specific) aspects that tend to affect only a limited set of joint points, while other styles of aspect oriented programming encourage more broadly scoped pointcuts, which naturally register more visibly on coupling metrics.

C. The Pointcut-Advice Churn Metric

To better detect F1 faults, we propose a new metrics, termed *pointcut-advice churn*, derived from the traditional churn measure, but adapted in two key respects. First, our new metrics is *targeted* in that it only considers the pointcut expressions of an aspect plus an advice signature, excluding any other code. Second, our new metric is *fine-grained* in that it works at the level of the tokens encountered rather then lines. The precise definition of the pointcut-advice churn metric is given in eq. 1, where a is an aspect, d runs over the advices of a, tokens_d^+ is the set of tokens added to advice d, and tokens_d^- is the set of tokens removed from d.

$$\mathsf{Pcut\_Adv\_churn}(a) = \sum_{d \in adv(a)} \left(|\mathsf{tokens}_d^+| + |\mathsf{tokens}_d^-| \right) \tag{1}$$

By measuring the number of clauses (tokens) used to determine when an advice should execute, and how what data should be extracted from the base code, this metrics aims to detect complex changes to pointcuts that might



(a) Pointcut(F1), Advice(F3) and all Faults VS Churn.

(b) Pointcut(F1), Advice(F3) and all Faults VS Pointcut-Advice Churn.

Figure 6. Comparing fault distribution relative to churn(left)and pointcut-advice churn(right)

indicate (i) the need to quantify “difficult to reach join points”, (ii) or high levels of data dependency, or (iii) both.

To check these intuitions we measured the correlation between pointcut-advice churn and faults (Table VI, with the coefficients for churn repeated for comparison). By contrast to churn, our new metric presents high correlation with F1 faults (0.85, to compare to 0.19 for churn). The combination of traditional churn and pointcut-advice churns also allows to clearly discriminate between the two types of faults, with F3 faults much more weakly correlated to pointcut-advice churn than to traditional churn (0.37 against 0.62).

Table VI
POINTCUT-ADVICE CHURN VS. CHURN

Metrics	#Faults	F1	F2	F3	F4
Churn	**0.55**	0.19	-0.01	**0.62**	0.14
Pointcut-Advice Churn	**0.77**	**0.85**	-0.04	0.37	-0.03

Another representation of the same data can be obtained by plotting the proportion of faults reached when selecting a certain amount of churn, starting with the modules with highest churn (Figure 6). This representation is similar to *receiver operating characteristic* (ROC) curves used in signal processing and visually illustrates how proportional faults are against a particular metrics, across modules. A purely proportional behaviour will show as a diagonal line, whereas an over-proportional behaviour (higher values contributing more to faults) will appear as a left-corner curve.

The difference between F1 and F3 faults, and between the churn and pointcut-advice churn metric is clearly visible on the ROC curves of Figure 6. Whereas the overall number of faults oscillate around the diagonal for both metrics, the behaviour of F1 and F3 faults is directly inverted between the churn (6-a) and pointcut-advice churn curves (6-b). In particular, F1 faults are clearly concentrated in modules with high pointcut-advice churn values, in a manner that is over-proportional, demonstrating the additional value of this metrics to analyse fault-prone pointcuts.

V. Related Work and Study Limitations

As emphasised in the previous sections of this paper, knowledge about how faults are introduced into aspect-oriented programs is still limited. Having this in mind, we next summarise pieces of work that we believe are mostly related to the research presented herein. They are distributed in two categories: (i) fault prediction based on software metrics; and (ii) the characterisation of faults in the context of aspect-oriented programs. In the sequence, we discuss some threats to the validity of the achieved results.

Fault prediction based on metrics: Browsing the literature enables us to find a plenty of research on using metrics as indicators of fault-prone modules and failures in software. For example, Nagappan and Ball [17] presented the results of an empirical study which evaluates the relationship amongst software dependencies, code churns and post-release failures observed in a large-scale software system. Based on the degree of dependence and on the collected code churns between two releases of the system, Nagappan and Ball built prediction models for post-release failures. The statistical analysis showed that the models have good accuracy in predicting post-released failures, therefore identifying system modules that should be given more attention, for example, during the testing and code inspection activities. In our study, particularly limited by the size of our data set, we only collected churn-based values to better understand the relationship between faults and AOP-specific mechanisms. We also proposed a fine-grained, churn-based metric for pointcut and advice signatures, which showed a good correlation of related faults in the analysed system.

More recently, Zimmermann and Nagappan [35] performed an experimental study to evaluate the effectiveness of network analysis on dependency graphs in predicting faults. The target system is the same large-scale software system analysed by Nagappan and Ball [17]. The results show that network analysis on dependency graphs outperforms complexity metrics in terms of recall and precision when predicting critical modules.

In particular for AO software, in our previous research we investigated coupling metrics as predictors of fault-prone modules in AO programs [10, 9]. At a first stage, we analysed the effectiveness of coupling metrics as indicators of fault-proneness in AO systems [9]. Faults were collected from several releases of a real-world AO system and used for the comparison of metrics for coupling and other internal attributes (e.g. depth of inheritance and weighted operations in module). We also computed a novel metric that quantifies specific coupling-related dependencies in AO software. The results showed that coupling metrics, which are not directives of object-oriented metrics, tended to be superior indicators of fault-proneness. This motivated us to further develop fine-grained coupling metrics for AO systems [10]. Based on a larger dataset, we evaluated an exploratory coupling metrics suite with respect to their capability of indicating fault-prone modules. Again, the results showed that a particular set of fine-grained directed coupling metrics – the Aspect Coupling metrics – has the potential to help create better fault prediction models for AO programs This study allowed us to gain insights into different styles of AOP. For instance, Telecom had a greater proportion of functional aspects such as `Billing` and `Timing`. In addition, the focus of Telecom study was designed around maintenance tasks to core language mechanisms such as pointcuts and advice whereas previous studies focused on refactoring from OO to AO implementations. A notable difference was that the average number advice-base code couplings was much higher in our previous study analysing three larger systems [36, 10] where faulty modules typically experienced a much higher variety and number of coupling connections. On the other hand, this work had a lower range (between 4 and 12) which may be a reason to the lack of correlation in the finer-grained coupling metrics such as Aspect Coupling.

Fault characterisation for AO software: Several authors have investigated the characterisation of AOP-specific fault types [6, 7, 37]. In general, harmful faulty-scenarios are described based either on simple AO programs or on the researchers' expertise. In a recent paper, Ferrari et al. [8] defined a fault taxonomy for AO software, which encompasses the fault types earlier defined by other authors. The taxonomy includes the four fault categories discussed in Section III and was preliminarily evaluated through the categorisation of a fault set identified from several AO systems in their previous research [36]. In total, 104 faults were analysed and classified. Besides that, the authors characterised the most recurring faulty implementation scenarios observed in the analysed systems, within each fault category. Differently from Ferrari et al. [8], in this paper we investigated how different implementation strategies are prone to introduce faults in the programs (e.g. reusing pointcuts or creating new ones, or directly or indirectly accessing context data). Besides that, we analysed the ability of coupling and churn metrics to indicate fault-prone modules.

In regard to the study limitations, using AspectJ can be pointed out as a constraint in our experimental evaluation and conclusions. In fact, as observed by Filman and Friedman [38] at early stages of research on AOP, other programming techniques (e.g. Intentional Programming, Meta-Programming and Generative Programming) are able to realise the concepts of AOP. On the other hand, AspectJ has been far the most investigated AOP language, upon which several facets of AOP have been developed and evaluated.

Another limitation of this study regards the size and representativeness of the evaluated system, in turn limiting the generalisation of the results. On the one hand, Telecom is indeed a small AO application that does not reflect the industrial practice with respect to complexity in terms of lines of code. On the other hand, as emphasised in Section III, Telecom is a well-known application which is distributed together with the AspectJ language [19]. This enabled us to reduce the effect of extraneous variables and to perform the experiment in a prespecified period of time.

The selection of maintenance tasks itself, to be performed by the study participants, may be considered a threat to the construct validity. The participants were given the freedom to modify different modules according to their own design decisions. Nevertheless, this allowed an in-depth comparison of the fault-proneness of a variety of implementation strategies.

VI. Conclusions

This paper presented the results of an exploratory study whose objective was to investigate how faults are introduced in AO programs during typical maintenance tasks (e.g. changing an existing feature or adding a new one). Eight pairs or experienced AOP developers were given the list of tasks to be performed as well the freedom to modify different modules according to their own design decisions.

We designed a test suite based on the functional requirements of the target system. It enabled us to identify and document faults from the several implementations. We then collected a variety of metrics, mostly different in nature (e.g. coupling, code churn and size), in order to understand how faults were introduced during the maintenance tasks as well the impact of changes in software dependencies on the correctness of the system during its evolution.

The results showed certain implementation strategies to be more fault-prone than others such as specific techniques for accessing data from base code modules and binding advice to pointcuts. In addition we compared the effectiveness of existing AO churn and coupling metrics to detect faulty modules and propose future directions for metrics in order to improve their accuracy at fault localisation. This work demonstrates the importance AO-specific fault categories to analyse how faults are introduced in AO software to aid the teaching, use and evolution of AOP techniques.

Our future research is motivated by capturing the overlooked attributes of fault patterns in new metrics. To check the usefulness of the newly proposed pointcut churn metrics, we also plan to apply it to larger systems, for instance, the ones evaluated in our previous research on fault-proneness of evolving AO programs [10, 36].

Acknowledgment

The authors received the following financial support: *Rachel Burrows*: UK EPSRC grant; *Fabiano Ferrari*: FAPESP (grant 05/55403-6); *Alessandro Garcia*: FAPERJ (grant E-26/102.211/2009), CNPq (grants 305526/2009-0 and 483882/2009-7), CAPES/PGCI (project 5688-09-4), and PUC-Rio (productivity grant).

References

[1] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. Lopes, C. Maeda, and A. Menhdhekar, "Aspect-oriented programming," in *ECOOP'97*. Springer, 1997, pp. 220–242.

[2] K. El Emam, W. L. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

[3] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE TSE*, vol. 31, no. 10, pp. 897–910, 2005.

[4] R. Subramanyam and M. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE TSE*, vol. 29, no. 4, pp. 297–310, 2003.

[5] A. B. Binkley and S. R. Schach, "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures," in *ICSE'98*, 1998, pp. 452–455.

[6] R. T. Alexander, J. M. Bieman, and A. A. Andrews, "Towards the systematic testing of aspect-oriented programs," Dept. of Computer Science, Colorado State University, Fort Collins/Colorado - USA, Tech. Report CS-04-105, 2004.

[7] M. Ceccato, P. Tonella, and F. Ricca, "Is AOP code easier or harder to test than OOP code?" in *WTAOP'05*, 2005.

[8] F. C. Ferrari, R. Burrows, O. A. L. Lemos, A. Garcia, and J. C. Maldonado, "Characterising faults in aspect-oriented programs: Towards filling the gap between theory and practice," in *SBES'10*. IEEE, 2010, pp. 50–59.

[9] R. Burrows, F. C. Ferrari, A. Garcia, and F. Taïani, "An empirical evaluation of coupling metrics on aspect-oriented programs," in *ICSE WETSoM Workshop*. ACM, 2010, pp. 53–58.

[10] R. Burrows, F. C. Ferrari, O. A. L. Lemos, A. Garcia, and F. Taïani, "The impact of coupling on the fault-proneness of aspect-oriented programs: An empirical study," in *ISSRE'10*. IEEE, 2010, pp. 329–338.

[11] M. Harman, F. Islam, T. Xie, and S. Wappler, "Automated test data generation for aspect-oriented programs," in *in AOSD*. ACM, 2009.

[12] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero, "Control and data flow structural testing criteria for aspect-oriented programs," *Journal of Systems and Software*, vol. 80, no. 6, pp. 862–882, 2007.

[13] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs," in *ICST'08*. IEEE, 2008, p. 52-61.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ," *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, 2001.

[15] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE TSE*, vol. 20, no. 6, pp. 476–493, 1994.

[16] M. Ceccato and P. Tonella, "Measuring the effects of software aspectization," in *WARE Workshop*, 2004.

[17] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM'07*. IEEE, 2007, pp. 364–373.

[18] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *ISSRE'10*, 2010, pp. 309–318.

[19] The Eclipse Foundation, "AspectJ documentation," Online, 2010, http://www.eclipse.org/aspectj/docs.php - 21/01/2011.

[20] Y. Coady and G. Kiczales, "Back to the future: A retroactive study of aspect evolution in operating system code," in *AOSD'03*, 2003, pp. 50–59.

[21] R. Laddad, "Aspect-oriented programming will improve quality," *IEEE Software*, vol. 20, no. 6, pp. 90–91, 2003.

[22] M. Mortensen, S. Ghosh, and J. M. Bieman, "Aspect-oriented refactoring of legacy applications: An evaluation," *IEEE TSE*, 2010.

[23] H. Shen, S. Zhang, and J. Zhao, "An empirical study of maintainability in aspect-oriented system evolution using coupling metrics," in *TASE'08*. IEEE, 2008, pp. 233–236.

[24] M. Bartsch and R. Harrison, "An evaluation of coupling measures for AspectJ," in *LATE Workshop*. ACM, 2006.

[25] R. Burrows, A. Garcia, and F. Taïani, "Coupling metrics for aspect-oriented programs: A systematic review of maintainability studies," in *ENASE'09*. Springer, 2009.

[26] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, and A. von Staa, "On the reuse and maintenance of aspect-oriented software: An assessment framework," in *SBES'03*. Brazilian Computer Society, 2003, pp. 19–34.

[27] J. Zhao, "Measuring coupling in aspect-oriented systems," in *METRICS'04 (Late Breaking Paper)*, 2004.

[28] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. Dantas, "Evolving software product lines with aspects: An empirical study on design stability," in *ICSE'08*, 2008, pp. 261–270.

[29] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid, "On the impact of aspectual decompositions on design stability: An empirical study," in *ECOOP'07*. Springer, 2007, pp. 176–200 (LNCS 4609).

[30] M. Stoerzer and J. Graf, "Using pointcut delta analysis to support evolution of aspect-oriented software," in *ICSM'05*. IEEE, 2005, pp. 653–656.

[31] O. A. L. Lemos, I. G. Franchin, and P. C. Masiero, "Integration testing of object-oriented and aspect-oriented programs: A structural pairwise approach for Java," *Science of Computer Programming*, vol. 74, no. 10, pp. 861–878, 2009.

[32] M. Rinard, R. Sălcianu, and S. Bugrara, "A classification system and analysis for aspect-oriented programs," in *In Proc. 12th Symp. on the Foundations of Soft. Eng.* ACM Press, 2004, pp. 147–158.

[33] S. Katz, "Diagnosis of harmful aspects using regression verification," 2004.

[34] F. Munoz, B. Baudry, and O. Barais, "A classification of invasive patterns in AOP," INRIA, Research Report RR-6501, 2008.

[35] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *ICSE'08*, 2008, pp. 531–540.

[36] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Masiero, T. Batista, and J. Maldonado, "An exploratory study of fault-proneness in evolving aspect-oriented programs," in *ICSE'10*, 2010, pp. 65–74.

[37] J. S. Bækken and R. T. Alexander, "A candidate fault model for aspectj pointcuts," in *ISSRE'06*. IEEE, 2006, pp. 169–178.

[38] R. E. Filman and D. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Aspect-Oriented Software Development*. Boston: Addison-Wesley, 2004, ch. 2, pp. 21–35.