Preprint of paper accepted by IEEE Transactions on Software Engineering, 2010.

# Aspect-Oriented Refactoring of Legacy Applications: An Evaluation

Michael Mortensen, Sudipto Ghosh, *Member, IEEE Computer Society*, James Bieman, *Senior Member, IEEE*

Department of Computer Science, Colorado State University

**Abstract**

The primary claimed benefits of aspect-oriented programming (AOP) are that it improves the understandability and maintainability of software applications by modularizing crosscutting concerns. Before there is widespread adoption of AOP, developers need further evidence of the actual benefits as well as costs. Applying AOP techniques to refactor legacy applications is one way to evaluate costs and benefits. We replace crosscutting concerns with aspects in three industrial applications to examine the effects on qualities that affect the maintainability of the applications. We study several revisions of each application, identifying crosscutting concerns in the initial revision, and also crosscutting concerns that are added in later revisions. Aspect-oriented refactoring reduced code size and improved both change locality and concern diffusion. Costs include the effort required for application refactoring and aspect creation, as well as a decrease in performance.

## I. INTRODUCTION

Aspect-oriented programming (AOP) [1] provides a new construct, an aspect, to modularize crosscutting concerns in code. AspectJ-like AOP languages expose identifiable execution points in a system, also called join points, at which program execution can be augmented or altered [2]. There are two types of crosscutting concerns, dynamic and static. Dynamic crosscutting modifies system behavior by augmenting or replacing the core program execution flow in a way that cuts across modules. Static crosscutting makes modifications into the static structure (e.g., the classes and interfaces) of the system.

Typically an aspect encapsulates constructs such as *advice*, *pointcuts*, and *inter-type declarations*.

- Advice is functionality that is executed when an exposed join point is reached. Advice can be specified as before, after, or around advice: *before advice* executes before the join point, *after advice* executes after the join point, and *around advice* executes instead of the join point but can execute the original join point. The aspects are *woven* into the primary code by a preprocessor, compiler, or run-time system.
- A pointcut is a set of join points specified by a pointcut expression.
- Inter-type declarations in AspectJ and introductions in AspectC++ allow the developer to modify a program's static structure by adding new methods or attributes to a class, or by changing the inheritance or interface implementation relationships between classes.

Aspect-oriented refactoring is the process of refactoring a application by moving code that implements crosscutting concerns into aspects. This process is intended to improve design structure by modularizing crosscutting code concerns [3]. Various researchers state that using aspects will result in better understandability and maintainability of the application [1], [3], [4], [5]. Maintenance of legacy applications consumes more time and resources than any other part of the software lifecycle [6], and thus, developers may want to adopt techniques that reduce maintenance costs. Another possible benefit of using AOP is that implementation of crosscutting concerns shows a high degree of variability and is often faulty [7], and using AOP would implement these concerns in a consistent manner.

This research evaluates the benefits and costs resulting from refactoring existing legacy software with aspects. The primary focus is on evaluating benefits related to maintainability, though we also investigate consistency issues.

We aim to provide a balanced view of the relative merits and challenges of using aspect-oriented refactoring in practice. We used the "Goal, Question, Metric" (GQM) approach [8] when we selected metrics to be used in our study. Our study has two main goals:

- *Evaluate the effects of aspects on program maintainability.* Replacing a concern that is scattered in many code locations with a single aspect can potentially reduce the number of changes during maintenance. For example, features such as caching and tracing typically require code to be scattered across many classes in many files, but this scattered code can be modularized as a single concern in one file using aspects.
- *Evaluate the cost factors of adopting AOP, including performance, testability, and defect introduction.* There are one-time costs from restructuring a program to use aspects, including the cost of creating the aspects and removing the scattered code that is replaced by aspects. Aspect pointcuts may be fragile, relying on naming conventions or program structures that can change over time. Maintenance changes that conflict with constructs used by pointcuts may introduce defects [9].

Prior studies of costs and benefits of aspect-oriented refactoring focus on systems created as research projects [10], on only one revision [10], [5], [11], or on a preselected set of aspects [7]. Here are some examples.

- Coady and Kiczales [5] evaluated aspects and modularity by refactoring four crosscutting concerns as aspects in revision 2 of FreeBSD. They reported several change metrics, such as the number of source code locations, functions, files, and sub-directories that required modifications in the original and refactored versions of the code.
- Kulesza et al. [10] created two versions of the same design as part of their research, where one is a Java-based object-oriented application and the other uses Java and AspectJ. To create a second revision, they carry out a set of maintenance tasks, comparing the changes made in the two applications.
- Figueiredo et al. [12] studied the benefits of AOP in the context of software product lines – applications that are deployed on different hardware platforms. They consider two product lines and evaluate several revisions. Although they also evaluate modularity and find that aspects provide benefits during maintenance, their focus is on *design* stability of the product lines – how change impact, modularity, and dependencies between components are affected when providing variations of the same application for different hardware targets (where some features are enabled and others are disabled).
- Ceccato and Tonella [13] extended the object-oriented metric suite of Chidamber and Kemerer [14] for aspect-oriented software. Tsang, Clarke, and Baniassad [15] extended the Chidamber and Kemerer metrics and compared two real-time applications, one created using real-time Java extensions and the other created with Java and AspectJ. No refactoring was involved.
- Hoffman and Eugster [16] refactored three Java applications, measuring coupling, cohesion, size, concern diffusion, and the number of reusable operations. Only one revision was considered.
- Bartsch and Harrison [11] created two online shopping applications, and had groups of subjects perform maintenance tasks on one of two separate applications, which were intended to be equally difficult to modify.

Our study is unique in several respects:

- We used three legacy industrial applications in our study, including one that had 51,600 lines of code in its initial revision. Thus, we could investigate crosscutting concerns that are actually present in deployed applications and track real changes that were made during software evolution.
- We follow the evolution of the applications across several revisions. We used 6 revisions in one application, 7 in the second, and 3 in the third. We obtained more data points and made more detailed observations this way. For example, the benefits of using refactoring may show up only after a few revisions. If we used two revisions, we would only observe the up-front cost of using the new technique without seeing the benefits later. Using more revisions gives us better opportunities to observe what changes were made to the aspects and to the primary code in subsequent revisions.
- Instead of using a pre-selected set of aspects, we used all the crosscutting concerns that we identified in the initial version of the application as well as those that were added in later revisions. This makes the study more realistic and recognizes that crosscutting concerns can change during the evolution of software. However, we did not restrict ourselves to change-prone crosscutting concerns because in a real development environment, developers may not know in advance which aspects will be more change-prone, and thus may not be selective in modularizing crosscutting concerns as aspects.

  Because the three applications were developed in the same application domain, some of the identified aspects were used in more than one application. For those aspects, we compare the benefits (e.g., code savings) in each application.
- Maintainability is a quality that often can only be measured through surrogate measures. Some studies [13], [15], [16] used internal product metrics, such as cohesion and coupling, which affect maintainability. However, coupling and cohesion measures for aspects still need further refinement and validation. Also, there is not a consistent way to compare the values of aspect-oriented coupling and cohesion to those of object-oriented coupling and cohesion. We used measures that directly affect the maintainer, such as lines of code, and number of methods, classes, and files that change. These correspond to the activities that the maintainer must undertake during software maintenance.
- We also classify real defects that we encountered during the study. These defects are either likely to be prevented or likely to be introduced through the use of aspects. We also describe the issues related to coverage measurement when we use existing regression tests during refactoring.

In previous work we described three of the 14 aspects used in this study [17], [18], [19], [20], and our mock systems-based refactoring approach [21]. This paper describes the remaining 11 aspects and evaluates the effects on maintainability of all 14 aspects across multiple revisions of the three applications.

The remainder of this paper is structured as follows. We present our pilot study approach in Section II. In Section III we present research questions and measures. In Section IV we describe the identified aspects used in our approach. Section V gives the results of both pilot studies. We provide additional discussion in Section VI.

Section VII summarizes related work. Finally, conclusions and future work are described in Section VIII.

## II. Approach

The study subjects are three proprietary VLSI CAD applications of Hewlett-Packard: `InstanceDrivers`, `PowerAnalyzer`, and `ErcChecker`. All are C++ applications based on an object-oriented C++ framework developed at Hewlett-Packard. The first author of this paper was also involved in the development and maintenance of these applications. Thus, we had someone with domain knowledge, in-depth knowledge of the code, as well as the expertise in aspect-oriented programming in the study team.

The framework contains classes and method calls for use in the VLSI CAD domain to provide basic functionality (e.g., read circuit files and create graphs of connectivity) so that applications can focus on specific tasks (e.g., analyze power consumption and identify certain types of circuits). Because the applications and framework were implemented in C++, we used AspectC++[1]. We refactored five crosscutting concerns as aspects in `InstanceDrivers`, eight in `PowerAnalyzer`, and seven in `ErcChecker`. There are 14 distinct aspects in all.

The `InstanceDrivers` application identifies the instances (transistors) that 'drive' electrical nets, providing electrical current. The initial version consists of 1,600 lines of code (LOC) and the final revision has 3,300 LOC.

The `PowerAnalyzer` application estimates power dissipation of electric circuits. It consists of 3 smaller tools and a library of common code that the tools use. The initial version contains 13,900 LOC and the final revision has 16,600 LOC.

The `ErcChecker` implements electrical circuit checks, such as checking for proper transistor ratios between the pull-up and pull-down transistors of an inverter, checking for fan-out limits, and checking for drive strength problems. In order to understand and fix the problems, the circuit designer needs access to contextual circuit data, which the `ErcChecker` displays and writes to a log file. The application has 51,600 LOC in the initial revision and 64,400 LOC in the final revision.

Table I shows overall metrics for each application.

TABLE I

Overall Metrics.

| Metric | `InstanceDrivers` | `PowerAnalyzer` | `ErcChecker` |
|---|---|---|---|
| Number of classes | 17 | 29 | 114 |
| Number of files | 25 | 86 | 204 |
| Max DIT (depth of inheritance tree) | 1 | 1 | 2 |
| Max number of children of a class (breadth) | 3 | 1 | 58 |
| Cyclomatic complexity (linearly independent paths) per module | 15.2 | 22.6 | 70.8 |

[1] `http://www.aspectc.org/`

### A. Refactoring approach

We use the following steps [21] to refactor the original applications:

1) Identify a crosscutting concern that can be refactored as an aspect.
2) Implement the aspect.
3) Remove code corresponding to crosscutting concerns from the application and weave the aspect.
4) Conduct integration testing of the refactored application by executing the regression tests.

*1) Aspect identification:* Aspects can be identified through several means, including aspect mining tools, clone detection tools, analysis of software design and architecture, manual inspection of source code, and developer knowledge of program structures. We identified aspects by finding similar, scattered code and searched for potential aspect idioms described in the aspect-oriented programming literature, such as the use of maps or sets for caching, and error handling code that calls `exit()` when system calls fail. While this is a systematic approach to identifying aspects, we recognize the limitations of the approach. The list of aspects we identified is, by no means, complete, and one could possibly identify more aspects using other approaches. For example, there are other techniques for identifying aspects, such as selecting to modularize modules within an application that are expected to be change-prone. However, our approach gave us an adequate number of aspects to conduct the study. We included all the aspects that we found irrespective of whether they would be change-prone or not. The rational was that developers in general may not be selective in choosing crosscutting concerns to modularize as aspects, in part because they may not know in advance which ones are more change-prone.

We used both manual inspection of code and a tool we implemented to identify potential aspects. This tool tokenized the lines of code that immediately followed each function call, creating a list of words (variable names, function and method names, and keywords) that followed the function call. The list of words was sorted alphabetically into a word set. The tool compared the word sets that followed a function call to the word sets that followed that same function call in other program locations. Function calls that had word sets with a high percentage of common words were flagged as potential aspects. The tool identified crosscutting concerns, such as error-handling, which were implemented with a function call (e.g. `fopen()`) followed by code that checked the return value of the call. Since code that followed `fopen()` was similar throughout the application, the word sets were identified as similar and the error handling code was refactored as an aspect.

*2) Aspect implementation:* In the second step, we use a test driven approach to implement each aspect. Long compilation and weave times, and the lack of an appropriate testing methodology for aspect-oriented programs are two main challenges here. When developers first create an aspect, they need to test it to make sure it has the correct functionality and strength [22], so that it matches all (and only) the desired join points in the application. Developers often experiment with several alternative aspect pointcut specifications and advice to test that the aspect correctly modularizes a crosscutting concern. The problem with unit testing aspects is that they cannot be tested in isolation. They must be woven with a primary program before they can be tested. Weaving and compiling an aspect with large legacy code takes a long time, which makes iterative changes difficult. With the systems we used,

it took up to 25 minutes to weave and compile with the real systems [21]. We address this problem using our mock system based approach. It is much easier and quicker to perform unit testing on the aspect by weaving it with a small mock system. In our study, it took less than a minute to weave and compile with the mock systems.

The mock system mimics the program structures in the application (or real system) that the aspect must interact with. Mock systems contain a small `main()` function, as well as class and function stubs to provide targets for pointcuts and to provide just enough structure and functionality for the advice to interact with. In contrast to the application, which may have thousands of lines of code, mock systems often contain fewer than 100 lines of code.

We create a mock system for each aspect. We often use a different mock system to test a new aspect because the aspect may require different structural and behavioral characteristics of the real system to be simulated in the mock system. We weave the aspect with the mock system, which also executes advised methods so that we can test advice behavior. Our previous work [21] provides guidelines for creating mock systems and describes the cost of creating them. Here we summarize the key guidelines:

- For spectator aspects defined by Clifton and Leavens [23] as aspects that do not change the behavior of advised modules (e.g., logging), we create method stubs with naming conventions such that the pointcut will match join point in the mock system and the real system.
- For non-functional aspects, we create mock class methods with the same parameter types and return types as the methods in the real system, but with simpler functionality.
- Identify components methods and classes in the real system that provide essential functionality in the mock system. Import the necessary components, such as by including a header file and linking against a system or framework library.
- Emulate the callsite context for join points using a similar number of arguments and argument types (e.g., simple scalar types, user-defined types, pointers, and templatized types such as STL containers). This is to identify and test changes to the control flow such as exceptions, method calls, and recursive calls.
- Import or emulate system functionality used by advice in the mock system.
- Use annotations in the mock system to check potential pointcut strength faults.
- Emulate aspect-aspect interactions when multiple aspects may advise the same join point.

We used various types of pointcut specifications depending on the nature of the join points that the aspect would advise. For aspects that advise many functions without a common naming convention, we used a list of function names as the pointcut. For aspects that advised all methods of the certain class, we used the class name and the type of the first parameter of the method as the pointcut pattern. To advise C++ functions, we refactored them into static methods of a class so that we could use the class name in the pointcut specification.

In general, pointcuts were chosen to match the join points for a refactored cross-cutting concern. In some cases, a concern was inconsistently implemented in the original application (e.g., checking for null pointers) and we wanted to apply the aspect consistently to all the appropriate locations. In these cases, we made the pointcut specification more general to include all such locations. We used our own judgement to decide what would be a consistent use of aspects. Since we were working only with the code and did not have access to design documents, we had no

way of knowing if certain cases of inconsistent uses were intentional.

After developing the aspect, we refactor the application to use the aspect and then test the refactored application using pre-existing regression tests. There is a possibility that all the crosscutting concern code is not completely removed from the original application. This could happen with any refactoring, such as extract method [24]. One might extract a new method but forget to call it from all the places where similar code was present. This may not cause the program to fail. However, with aspects, one runs into another problem. If the code is not removed but the aspect still gets woven because the pointcut matches the join point, then the same execution can occur twice, potentially leading to program failure. Thus, it is important that the refactored system be tested.

### *B. Case study approach*

No matter what approach is used for aspect-oriented refactoring, there will be costs related to identifying aspects, creating aspect code and changing primary code. When a mock system-based refactoring approach is used, there are additional costs related to developing mock systems and performing unit testing. In this paper, our focus is on the cost of aspect code creation and primary code changes. We do report the cost of developing mock systems but the cost of identifying aspects is outside the scope of this paper.

Each application has a source code repository that contains the version history of each source file. For each application, we start at revision 1, and identify and refactor crosscutting concerns as aspects so that we have revision 1-aop (refactored version of revision 1). We report the cost incurred when refactoring the first revision of an application. We compare the original and refactored code for any revision of the software, using metrics such as size, execution time, memory usage, and test coverage.

Next, we examine the changes made between the current revision (i.e. revision 1) and the next revision (i.e. revision 2), making the functionally equivalent changes in refactored revision 1 (i.e. 1-aop) to create revision 2-aop. Some of the changes made between revisions may also correspond to crosscutting concerns that could be refactored as aspects. Thus, in addition to the aspects identified in the initial revision, aspects may also be identified based on the changes occurring after the first revision. We compare change-related metrics when going from revision `N` to revision `N+1` on the original and refactored branch of the code. We follow this process over several revisions so that costs and benefits can be measured in terms of long-term maintenance. We restrict our study to the aspects that we identify based on our approach described in Section II-A.

Figure 1 shows the process. The rectangles represent revisions and the label, “Original changes”, between revisions represent the changes that were made in the original application. Corresponding changes are made in the corresponding revisions of the refactored version.

## III. Questions and Measures

With the overall goal of evaluating the cost and benefits of aspect-oriented refactoring, we identified eight key research questions as described below. To answer each question, we selected metrics to collect during the study.

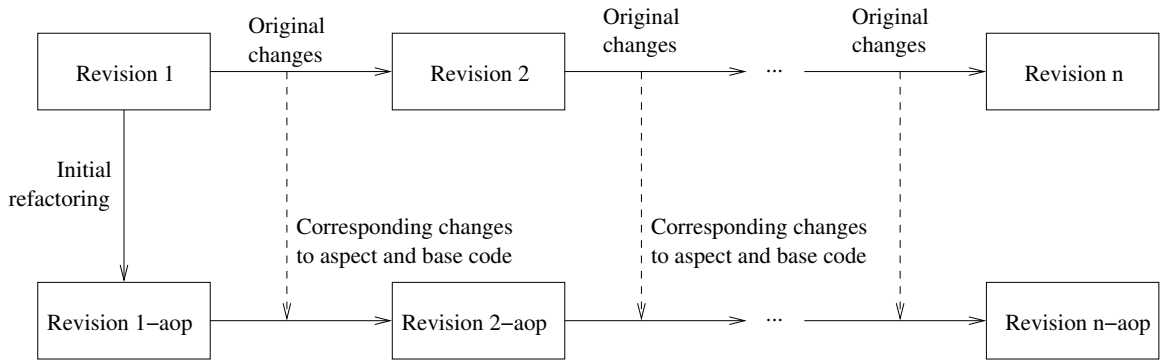


Fig. 1. Evaluating refactoring using revision history

### A. What is the cost of creating an aspect?

We measure direct costs of creating the aspect, such as time, and also measure code size because larger size suggests more time required for implementation, testing, and maintenance. Measures include the size of the mock system, the size of the aspect, the time spent developing the aspect, and the number of iterations (creating or modifying the mock system, creating or modifying the aspect, weaving, and testing) that were required until we were satisfied that the aspect's pointcut and advice were correct.

### B. How much code is changed when refactoring to use aspects?

We measure the number of lines changed in the original application source code. Lines of code changed is a surrogate for cost; it represents the cost of time spent understanding, adding, or deleting code. We differentiate between modifications of source code and deletions of source code. Modifications may include renaming functions or removing conditional checks of system calls without removing the system calls themselves. For example, we might rename certain methods to follow a naming convention (e.g., to start with a common prefix) so that an aspect pointcut can match methods that use that convention. Deletion of code occurs when code is modularized into an aspect, and is typically simpler than modifications, which represent changes needed to support added aspects. Figueiredo et al. [12] also report the specific types of modifications (deletions, additions, and changes) made during aspect-oriented refactoring.

### C. Does the use of aspects reduce the amount of source code that must be maintained?

Refactoring should reduce the amount of source code in the application when scattered, often duplicated, code is modularized to a single aspect. The reduction occurs because some code is deleted during initial refactoring and some code addition is avoided in subsequent revisions of the crosscutting concern that is implemented by the aspect. An aspect definition adds new source code to the application. Thus, for a reduction of total source code to occur, refactoring needs to delete more lines of scattered code than are required to implement the aspect.

We measure source code size in the original and refactored application, including aspect size. To measure the size of AspectC++ applications, we count the lines of the primary source code (classes and functions that developers write) as well as the lines in the aspects.

Kulesza et al. [10], Tonella and Ceccato [13], and Hoffman and Eugster [16] all compared the size of the original application to the size of the refactored application.

### *D. Do aspects increase memory usage and execution times?*

We use refactoring to improve maintainability rather than performance. However, serious performance degradation is not acceptable. Performance is a concern in CAD applications because they often use large data sets in memory and have long execution times.

Executable size can increase when aspects are used because the AspectC++ weaving process inserts a method call at each join point. The AspectC++ weaver generates a class in the woven code for each join point. This join point-specific class represents static and dynamic data that can be used by the aspect. Invoking a method and providing extra join point-specific data may increase memory usage and execution size, although Lohmann et al. [25] report low overhead when they used aspects in place of virtual methods. We report the size of woven code even though it is generated by a weaver and the developer does not directly maintain it. The woven code is needed only when statement coverage tools such as `gcov`[2] are used. Coverage reports refer to statements in the woven code, and thus, the developer must work with the woven code to reason about missing coverage.

We collected object size data because it may impact performance. To observe trends, we measure the performance of four application revisions. We also compare the memory usage and execution time of regression tests of the refactored and original applications.

Since AspectC++ is a research tool and not a mature product such as AspectJ, qualities such as performance and code size might not be representative of how a mature compiler or runtime environment would perform. Mature environments typically have more optimized implementations. Unfortunately, AspectC++ is the only known language for aspect-oriented programming in C++.

The increase in code size depends on the amount of advice that is woven into the system. Thus, we also report how many primary code locations are advised by each aspect. This join point data can help to understand code growth, memory usage, and performance penalty for an aspect.

Coady and Kiczales [5] and Lohmann et al. [25] also studied the run-time overhead of using aspects.

### *E. Are changes from one revision to the next more localized if crosscutting concerns are implemented as aspects?*

Hannemann and Kiczales [3] define change locality as the number of modules (functions, classes and aspects) that are changed for a particular revision. In theory, refactoring should improve change locality because scattered code is replaced with a single aspect. However, changes between revisions that are not related to crosscutting concerns

[2] `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

will not affect change locality. Moreover, just as some design patterns exhibit different levels of change-proneness in practice because of the types of design decisions they represent [26], some kinds of aspects may be more change prone. Thus, we report the number of changes made to the aspects.

We measure the number of source code lines, modules, and files changed between revisions, comparing the refactored and original applications. Module change locality is measured by counting how many modules change between revisions, where a module is a function, method, or class attribute lists. Changes to a scattered concern in the original application will affect many modules. In contrast, the refactored application will only require a change in the aspect, thereby reducing the number of modules that change. File change locality is measured by counting how many files are changed between revisions. Refactoring should reduce the number of files changed between revisions because a concern is implemented using classes and functions in many files.

Our approach to measuring change locality in terms of locations and files is analogous to the approach of Coady and Kiczales [5].

### *F. Are concerns less diffused in source code when aspects are used?*

Concern diffusion indicates the separation of concerns in source code. Reducing concern diffusion leads to code that focuses on a single concern instead of frequently switching between concerns. Kulesza et al. [10] define *Concern Diffusion over LOC* (CDLOC) as the number of transition points for a concern in the source code. They refer to transition points as *concern switches*, where the code transitions from one concern to another. For example, consider the following sequence of code, which has two concern switches:

```
circle.getColorInfo();
if(enableLogging) { ... logging code here }
circle.draw();
```

The first concern switch occurs when switching to the logging concern, and the second one when switching back to the primary code concern.

Kulesza et al. [10] compute CDLOC for the entire program by associating each line of the program with a particular concern. We do not compute CDLOC for the entire program. Instead we compute the reduction in CDLOC after performing refactoring by counting the number of concern switches reduced. Continuing the example above, if the line that performs logging between calls to the circle class is replaced by an aspect, then where there were two concern switches, there are now zero concern switches—a concern switch reduction of two.

Since aspects modularize crosscutting concerns as a single module, the refactored version should have lower concern diffusion than the original.

Hoffman and Eugster [16] also measure concern diffusion, but focus on a single concern's diffusion rather than on concern switches.

### G. What effect does aspect-orientation have on testing, and in particular on test coverage?

Software development experts such as Fowler [24] recommend running regression tests whenever a refactoring step is performed. Thus, we investigate the challenges related to regression testing and coverage measurement when aspect-oriented refactoring is performed in an industrial setting. Performing unit tests and regression tests, and then evaluating and analyzing the coverage data represents a cost for performing refactoring.

We use available regression tests on the original and refactored applications. To highlight aspect-specific test coverage data, we measure *join point coverage* [17]. Join point coverage requires executing each join point that is matched by each aspect, focusing on testing each aspect in all of the contexts where it is woven. We developed tools [21] to gather information about coverage of advised join points. Using aspect-specific coverage information helps us check whether the aspect was woven in the right places. We cannot just rely on passing the regression tests because in general, no test suite can be guaranteed to test everything in a system.

The join point coverage criterion complements traditionally used criteria, such as statement coverage, which identifies the primary concern code and advice body code that is executed. We measure statement coverage using `gcov` on both the original code and refactored code. Statement coverage is a basic and naive form of coverage measurement but it captures the essential characteristics and there is tool support that is easy to use.

Existing test coverage tools such as `gcov` must be used on source code, which in our case, is the woven source code. Developers only care about coverage data for the code they have written and not for something that is generated by the weaver. This becomes a problem when AspectC++ and `gcov` are used because there is no way to separately view coverage of the original source code from the coverage data from the woven code. Thus, if there is woven code that isn't covered, the developer needs to spend more resources to investigate the cause for lack of coverage.

When we performed the study, we had access to the regression tests for `InstanceDrivers` but not `ErcChecker`. For the `PowerAnalyzer` we had access to only half the regression tests. The missing regression test cases lower the coverage. However, because our focus was on the refactoring, attempting to obtain the old test cases to construct new test cases was beyond the scope of this study.

### H. Does aspect-oriented refactoring increase or reduce defects?

Defects may be introduced during refactoring. Examples include incorrect changes to the primary code, incorrect pointcut definitions, and incorrect advice code [22]. Using aspects may help reduce defects as well. A scattered crosscutting concern (e.g., return code checking) that was inconsistently implemented in the original application may be consistently implemented by an aspect that advises all pointcuts and applies the same advice [17].

## IV. Summary of the concerns identified in the legacy applications

We identified five crosscutting concerns as aspects in the `InstanceDrivers` application: `Caching`, `CheckFwArgs`, `Excepter`, `Singleton`, and `Tracing`. We identified eight crosscutting concerns in the `PowerAnalyzer` application: `CadTrace`, `CheckFwArgs`, `Excepter`, `FetTypeChkr`, `FwErrs`, `Timer`, `UnitCvrt`, and `ViewCache`.

We identified seven crosscutting concerns as aspects in the `ErcChecker`: `Caching`, `CheckFwArgs`, `Excepter`, `ErcTracing`, `FwErrs`, `QueryConfig`, and `QueryPolicy`. Two aspects, `CheckFwArgs` and `Excepter`, are used by all three applications, and two others, `Caching` and `FwErrs`, are used by two applications. We created 14 aspects between the three applications.

Table II shows the applications associated with each aspect. The first column lists the aspect. A check mark in the next three columns shows that the aspect was used in the corresponding application. If an aspect was used in multiple applications, we use an asterix to indicate the application for which we created the aspect the first time.

TABLE II

ASPECTS AND SYSTEMS

| Aspect | InstanceDrivers | PowerAnalyzer | ErcChecker |
|---|---|---|---|
| Caching | | √ | √ * |
| CadTrace | | √ | |
| CheckFwArgs | √ * | √ | √ |
| ErcTracing | | | √ |
| Excepter | √ | √ * | √ |
| FetTypeChkr | | √ | |
| FwErrs | | √ * | √ |
| QueryConfig | | | √ |
| QueryPolicy | | | √ |
| Singleton | √ | | |
| Timer | | √ | |
| Tracing | √ | | |
| UnitCvrt | | √ | |
| ViewCache | | √ | |

*a) Caching:* The `Caching` aspect [19] is used in the `ErcChecker` to improve application performance by not recomputing derived values for circuit elements (e.g., total capacitance of an electrical node). `InstanceDrivers` uses caching to avoid reading configuration files and traversing design data more than once for instances of the same type of CAD cell. `Caching` was used in 27 locations of the final revision of `ErcChecker` and in one class of `InstanceDrivers`. Caching is implemented using around advice which returns a cached value if the data has already been computed. Otherwise, the advice proceeds with the original method call. Aspect inheritance allows a base caching aspect to be specialized for different call sites and data types in the primary code.

*b) CadTrace:* The `CadTrace` aspect provides tracing used to determine the location of an invalid framework method call. A framework method in the `PowerAnalyzer` called `exit()` after indicating that a null pointer had been encountered. Calling `exit()` limits visibility when using a debugger such as `gdb` because the program terminates without preserving any system state. By contrast, using `assert` also exits a program, but creates a core file with the state of the program so that the call stack and program state can be analyzed.

The error message listed the name of the framework iterator method name. However, the method was in a base

class that was inherited by several subclasses, so there were many candidate calls in the application that may have triggered the error. These calls represented possible locations of the defect and crosscut 18 locations in four files.

The prior approach to debugging such problems involved adding print statements around all calls that could have triggered the error. This required modifying all 18 locations in four files, finding the defect and fixing it, and removing the 18 modifications from the four files. This process is tedious and error prone. A single aspect can automatically provide the same tracing, using the framework iterator initialization as the pointcut. The advice uses AspectC++ features to print the context of each iterator call.

*c) CheckFwArgs:* The `InstanceDrivers` application contains a utility layer that implements application-specific algorithms for framework objects. This layer calls framework API methods using framework object pointers that are passed in as parameters to the utility layer functions. Object pointers should be checked to confirm that they are not null to prevent fatal run-time errors. Null pointer checking is performed in some of the functions in the original application. The `CheckFwArgs` aspect removes duplicate checking code and improves safety by automating this check consistently across all callsites.

We used template meta-programming and the AspectC++ reflection API to implement the aspect, so that it can handle all framework utility methods even though they have different numbers of arguments and use several different framework and non-framework types as parameters. The aspect uses around advice, which returns if a null object is found, preventing the use of a null pointer with framework methods. The pointcut advises a group of functions without enumerating each one. We grouped the functions into a class as static methods, had the class inherit from an empty mixin class (C++ allows multiple inheritance), and created a pointcut that specified all the methods of sub-classes of the mixin class. We chose the mixin name so that it would indicate that an aspect was advising this code, which communicates the relationship between the primary code and an aspect to developers in a less oblivious way, similar to annotation-based weaving. We also created a C++ macro that redirected existing calls to the original function names to the static class methods of the same name.

*d) ErcTracing:* The `ErcTracing` aspect provides detailed tracing for the `ErcChecker`, replacing scattered inconsistent code. We had already developed the `Tracing` aspect, but because the `Tracing` aspect uses class data rather than global variables, we did not use aspect inheritance. The `ErcTracing` aspect contains seven advice blocks for different components in the `ErcChecker`. Each advice block uses a different global variable to indicate when tracing that subsection of code should be enabled. There are minor differences between the advice bodies, but all seven reuse the library of template metaprogramming code from the `Tracing` aspect to print method arguments.

*e) Excepter:* The `Excepter` aspect replaces similar code that checks return values from functions such as `getenv()` and `fopen()`. These functions use the return code idiom [27], i.e., they signal errors through return codes. The `Excepter` aspect provides a more modular approach for consistently handling errors than the scattered checks [20]. It uses after advice to check the return value of each function call. If the return code is null, the advice throws an exception. The aspect contains a second advice that provides a try/catch block around `main`. The try/catch block catches the exception thrown by the first advice and exits. If developers want to catch the error in the function call itself instead of relying on `main` to exit, they must manually add a try/catch block around that

function call to catch the exception thrown by the first advice.

Both the `Excepter` and `CheckFwArgs` aspects provide error checking, which is an extra-functional concern. Such concerns relate to characteristics such as dependability, configurability, and performance [28].

*f) FetTypeChkr:* The `FetTypeChkr` replaces parameter checking that is done in the `FetType` class of the `PowerAnalyzer` application. The aspect uses the pointcut pattern-matching capability in AspectC++ to advise all methods of `FetType` that have an integer as the first parameter. Even though this aspect advises only one class, it modularizes a check across 14 methods and it ensures that methods added to this class in the future will also be advised. The aspect advice validates that the range does not exceed the maximum value for the class, which is stored as a class attribute. The aspect replaces scattered checks on several methods with a single advice body to perform the same checking, thereby modularizing pre-condition contract checking for the class. This aspect is similar to design-by-contract checking aspects that have been explored by several researchers [29], [30], [31].

*g) FwErrs:* The `FwErrs` aspect refactors crosscutting code for handling framework-specific failures. We identified this aspect based on idioms in code that detects these errors and handles system shutdown due to fatal framework errors. This aspect differs from the `Excepter` aspect, which is for non-framework system calls in C or C++ where the response to errors depends on the context of use. The `FwErrs` aspect, like the `Excepter` aspect, uses around advice to throw an exception when a return code from an advised method indicates an error. The refactored application relies on the aspect advice for `main` to catch the exception and exit.

*h) QueryConfig:* The `QueryConfig` aspect provides run-time configuration for the electrical queries. Because of the number of queries and the run-time of the tool (often several hours for a circuit), the `ErcChecker` allows users to turn any of the checks off through a user configuration file. Before time-intensive circuit traversal or electrical query evaluation, each of the 58 types of electrical calls a configuration method to see if the user has disabled that check.

Each query in the `ErcQuery` class hierarchy of `ErcChecker` calls `getQueryConfig()` before actually performing the checks implemented by the electrical query. The `QueryConfig` aspect uses around advice, ensuring that queries always check their configuration status before executing. An additional benefit of the aspect is that it can use reflection in the AspectC++ API when passing the current query name to `getQueryConfig` – the original C++ code embeds the class name as a literal string, since C++ lacks reflection.

*i) QueryPolicy:* The ErcChecker implements 58 different electrical checks as subclasses of an abstract base class `ErcQuery`. A static method in each class, `createQueries()`, calls a common set of virtual methods in a manner similar to the Template Method design pattern [32].

For each query, `createQuery()` performs the same six conceptual steps. The first three steps identify circuit data, create instances of query objects, and call the `executeQuery()` method. The last three steps add failing queries to a container class (`LevelManager`), write query data to a log file, and delete queries that did not find electrical problems. Because of algorithmic and circuit-specific differences, each query has its own `executeQuery()` implementation. The `QueryPolicy` aspect replaces steps 4 through 6 as after advice for the call to `executeQuery()`.

*j) Singleton:* The `Singleton` aspect [3] modularizes similar code for a class that has multiple constructors for various contexts, but which all trigger the creation and sharing of a single instance. This aspect uses around advice with the core concern's static creation methods to ensure that a constructor is only called when the global instance has not already been created.

*k) Timer:* A common practice in VLSI CAD software is to write time stamps to a log file before certain steps or write the elapsed time after certain steps. The `PowerAnalyzer` does this by implementing a `TimeEvent` class containing methods that are called before and after certain steps. The `Timer` aspect [18] automates the creation of a `TimeEvent` object, which is executed before and after calling the function, and used to record elapsed time.

The functions that need to be timed have various names. A pointcut that matches all the functions would need to list all the names. An alternative that simplifies the pointcut and indicates intent in the core concern is to rename the functions from `FunctionName` to `tmrFunctionName`. This allows the aspect to advise all functions beginning with `tmr`.

Some of the code with `TimeEvent` calls are part of a large procedural function that contains several sub-steps to be timed. We refactored the sub-steps using the Extract Method refactoring approach [24] using a function name beginning with `tmr`. Extracting and renaming methods required more changes than for the other aspects but results in less fragile pointcuts. We did not have to change the pointcut in the next revision as long as we renamed the functions to be timed with names beginning with `tmr`.

*l) Tracing:* The `Tracing` aspect refactors a concern commonly reported in the aspect-oriented programming literature – program tracing [33]. The `InstanceDrivers` application contains classes that have optional verbose output enabled via command-line arguments. The verbose mode causes method calls and their return values to be reported, as well as calls and return values of any functions called from within the traced methods. Besides being scattered throughout the class, such tracing code is often inconsistent. The `Tracing` aspect's around advice uses template-metaprogramming in order to process and print out parameter values and return values regardless of the number of parameters, parameter types, or return type.

*m) UnitCvrt:* For many scientific applications, units may be represented using different internal formats but must be written out with a standard unit of measure, such as nanometers or picofarads. The `UnitCvrt` aspect replaces scattered code that converts CAD property data into a standard unit of measure before printing the data in reports. The original code is in multiple functions across several files rather than in a single module.

The value to be converted is stored as a CAD property object, attached to an electrical net object. The challenge is that there are many other property objects stored in a similar way that do not need to be converted because they do not deal with units, but deal with other relationships. Only properties that store the length and width of a transistor need to be converted. Also, new code could be added to later revisions to access the named properties are based on unit values.

This aspect advises all property accesses by using around advice to intercept all calls to the `GetValue()` method of the `Property` class. The advice always calls `proceed()` so that the method is executed. Next, the advice calls the property's `GetName()` method and if required, the value is converted into a standard unit of

measure. By using an aspect, we guarantee consistent use of all values associated with this specific named property object.

*n) ViewCache:* The `ViewCache` aspect provides caching of different data views in the `PowerAnalyzer` application. We could not extend the `Caching` aspect [19] from `InstanceDrivers` to create the `ViewCache` because of differences in caching between the two applications. The `ViewCache` does not cache a value for a function or method call like the `Caching` aspect. Instead, it provides primary code methods with the capability to avoid reloading the necessary component views (e.g., schematic view, electrical view, and layout view) when those views already exist within the loaded framework data. The `ViewCache` does not refactor application-specific loading of component views performed during framework initialization, but accesses the framework data only to check what views are already loaded. The `ViewCache` aspect uses around advice to replace scattered code that determines if the different views are already loaded into memory during certain operations.

## V. Results

In the following paragraphs, we explain the results of our study using the three applications.

### A. Cost of developing aspects with mock systems

We show mock system size and cost measurements for the initial creation of each aspect in Table III. We report the time to create the aspects and mock systems, and to iteratively test and refine the aspects using the mock systems. If an aspect can be reused in more than one application, we do not repeat the mock system process, thereby decreasing the cost of future use of the aspect.

The `Caching` aspect required the largest mock system and took the most iterations because of the complexity of the caching functionality. Although the mock system was large, its creation time was less than for some aspects because it simply needed to contain different kinds of function and method calls to be cached, while other aspect mock systems needed to have stubs or mock components developed. It was created for the `ErcChecker`, which required caching of both procedural functions and object-based method calls. In addition, we created several specialized caches through aspect inheritance that can also track cache hit rates and cache usage to report performance [19]. We developed a set of four base aspects to implement four different caches. These aspects can be reused through aspect inheritance. The total lines of code for the four aspects is 130. The concrete realization requires only three lines of aspect code to select the cache and specify the pointcut. When the `Caching` aspect was reused in `InstanceDrivers`, it had 33 LOC (30 for the base aspect and 3 for the pointcut). The mock system was reused and thus, had no new costs.

The `ViewCache` aspect is a simplified type of cache. The experience with creating the `Caching` aspect enabled the rapid development of `ViewCache`; only one iteration was needed.

The `Singleton` aspect is also structurally similar to the `Caching` aspect. The `Singleton` aspect advises a static creation method and avoids executing it more than once. Our experience with the `Caching` aspect helped us create the `Singleton` aspect with just three iterations in 20 minutes.

TABLE III

ASPECT AND MOCK SYSTEM SIZE AND COST

| Aspect | Aspect LOC | Mock System LOC | Iterations to Create Mock System | Aspect and Mock System Creation Time (minutes) |
|---|---|---|---|---|
| Caching | 130 | 600 | 15 | 65 |
| CadTrace | 8 | 1 | 1 | 30 |
| CheckFwArgs | 29 | 76 | 3 | 50 |
| ErcTracing | 108 | 30 | 2 | 30 |
| Excepter | 28 | 125 | 5 | 50 |
| FetTypeChkr | 11 | 60 | 3 | 40 |
| FwErrs | 16 | 25 | 1 | 15 |
| QueryConfig | 14 | 20 | 3 | 30 |
| QueryPolicy | 10 | 160 | 10 | 60 |
| Singleton | 5 | 28 | 3 | 20 |
| Timer | 12 | 50 | 4 | 30 |
| Tracing | 62 | 150 | 8 | 90 |
| UnitCvrt | 26 | 122 | 5 | 75 |
| ViewCache | 4 | 33 | 1 | 15 |

The `Tracing` aspect took more time to create and more iterations than all but the `Caching` and `ErcTracing` aspect. The greater effort was due to the complex nature of the aspect, which uses C++ templates within the advice to check an arbitrary number of parameters. In addition, virtual methods were used so that the advice could print out objects of many different data types.

The `ErcTracing` aspect was created after `Tracing`. The mock system is larger, because the aspect contains seven advice blocks that advise different code components. Fewer iterations were required than for `Tracing` because of our prior experience developing `Tracing`.

We also used templates that call virtual methods to create the `CheckFwArgs` aspect. Since we had already used the technique for `Tracing`, we could create the advice faster for `CheckFwArgs`. However, developing the pointcut required several iterations because of the complexity of using template metaprogramming and mixins.

Developing the `Excepter` aspect required the third most iterations. The aspect must be woven with many functions that return several different data types (e.g., `bool`, `int`, and `int*`). In addition, we explored approaches for using aspects to locally catch exceptions for non-fatal cases [20].

The `FwErrs` aspect is similar to `Excepter`, which allowed us to develop `FwErrs` with fewer iterations. Since `FwErrs` advises just a few framework methods rather than several different functions, the mock system was also smaller.

For the `Timer` aspect, we created a mock system with function names beginning with `tmr`, and ensured that the advice correctly logged execution times of the functions. Several iterations were required to get the functionality

implemented correctly. The PowerAnalyzer had nested timed functions, which caused the early versions of our aspect code to incorrectly reset the start time for the outer method when invoking the nested method. We emulated the nested function structure in our mock system and made changes in the aspect to correctly handle this case.

The pointcut for the FetTypeChkr aspect uses the class name and the type of the first argument so that only methods whose first argument was of type int would be matched. The mock system required 60 lines, most of which were in a class with the same name as the one in the real application. We needed two iterations to correctly define the pointcut because we had not previously used pointcuts that matched both names and argument types. A third iteration completed the advice implementation.

Because the UnitCvrt aspect advises functions that use the property objects defined in the VLSI framework, its mock system uses code from the VLSI framework. The mock system creates several framework objects. Some of them have properties with names that indicate a unit conversion, and thus, must be advised by the aspect. To ensure that the aspect did not apply unit conversion to unintended objects, the mock system created objects with property names that did not indicate unit conversion, and also objects with no properties added. We defined the correct pointcut in the first iteration. However, it took four iterations to create advice that would correctly and efficiently convert units of properties. This was a challenge because the properties are stored as strings. The advice must convert the string to a floating point number. Rather than storing the converted value as a string, the advice cached the converted value for each object. The next time the program tried to access a property of the object, the value was retrieved from the cache.

The QueryPolicy and QueryConfig aspects modularize crosscutting code specific to the ErcQuery class, which is part of only the ErcChecker. In the mock system for QueryPolicy, we model the class hierarchy of the ErcQuery class. In addition, a driver file (main.cc) creates and executes query objects, storing results data internally. Once we had created the mock system for QueryPolicy, we were able to use it with only minor changes as the mock system for creating and testing the QueryConfig aspect.

Three factors influenced the amount of effort required to create and test an aspect using a mock system. First, aspects for which we needed to create larger mock systems tended to require more time to develop. We created larger mock systems when an aspect needed to be woven into more structures (e.g., several sub-classes of a common base class) or interacted with the application in a way that required more functionality than empty stubs. Mock system development time can be reduced by reusing system code when possible [21]. Second, the aspects vary in size and complexity. For example, because the Caching aspect used inheritance and cached values of methods and functions using a variety of data types, creating it took longer than FwErrs, which used no inheritance and checked the return value of the advised functions. At times, there is tension between aspect complexity and refactoring cost. For example, renaming methods to begin with tmr simplified the pointcut of the Timer aspect, but required more refactoring cost. By contrast, using mixins and inheritance with the pointcut added time to the development of the CheckFwArgs aspect, but reduced the amount of refactoring required. Third, gaining experience with features of AspectC++ reduced the time required for aspects that used those features. Creation times for the ViewCache, FwErrs, and Singleton aspects were all reduced because we became familiar with the AspectC++ features

needed.

### *B. Initial Refactoring Change Impact*

We consider the code added, modified, or deleted when the aspect is *first* introduced to the application. The `InstanceDrivers` data is shown in Table IV. The size of the aspect (LOC) is not a refactoring cost, but is shown to contrast with the changes made to the primary code for each aspect.

TABLE IV

INSTANCE DRIVERS: SOURCE CODE CHANGES MADE

| Aspect Name | Aspect Size(LOC) | Changes to primary code (LOC) | | | |
|---|---|---|---|---|---|
| | | Additions | Changes | Deletions | Total |
| Caching | 3 | 0 | 0 | 18 | 18 |
| CheckFwArgs | 29 | 13 | 7 | 0 | 20 |
| Excepter | 28 | 0 | 0 | 1 | 1 |
| Singleton | 5 | 0 | 0 | 11 | 11 |
| Tracing | 62 | 0 | 0 | 16 | 16 |
| Total | 127 | 13 | 7 | 46 | 66 |

For the `InstanceDrivers` application, 66 lines were changed, of which, 46 (more than half) were deletions. Four of the aspects required only deletions. The `CheckFwArgs` aspect is the only aspect that needed additions; its pointcut required that the advised functions be refactored as static methods of an advised class. The number of deletions is greater than the size of the aspect for two out of five aspects.

The `Caching` aspect uses aspect inheritance to reuse a more general caching aspect from a collection of abstract caching aspects that we had created for the `ErcChecker` [19]. The base aspect size is 30 LOC. However, because only 3 new lines of code were written in order to extend the base aspect, the table shows 3 LOC.

The `CheckFwArgs` aspect removed no lines because none of the checks it performs were implemented in the initial revision. These checks were not present in subsequent revisions of `InstanceDrivers`, so we did not realize any code savings. However, these checks are necessary since the other two applications had code to perform the same checks, as shown in Table V and Table VI.

The `Excepter` aspect allowed us to delete one line in the `InstanceDrivers` code because the error checking it provided was only implemented as a line check in one location in `InstanceDrivers`. Even though it deleted only one line, we used it because we had already implemented it for the `PowerAnalyzer`, and it might eliminate the need for more code in later revisions. Using the `Excepter` aspect allowed us to save 24 lines in three of the subsequent revisions. This is reflected later in Table VII, in which we see that code reduction using AOP increases with later revisions.

The largest aspect, `Tracing`, required more lines to implement than the number of lines removed (16). However, as reflected later in Table VII, the use of `Tracing` resulted in 66 lines being removed by the last revision.

TABLE V

POWERANALYZER: SOURCE CODE CHANGES MADE

| Aspect Name | Aspect Size(LOC) | Changes to primary code(LOC) | | | |
|---|---|---|---|---|---|
| | | Additions | Changes | Deletions | Total |
| Timer | 12 | 34 | 20 | 79 | 133 |
| Excepter | 28 | 17 | 57 | 112 | 186 |
| CheckFwArgs | 29 | 18 | 4 | 81 | 103 |
| CadTrace | 8 | 0 | 0 | 0 | 0 |
| ViewCache | 4 | 0 | 1 | 21 | 22 |
| FetTypeChkr | 11 | 0 | 0 | 18 | 18 |
| FwErrs | 16 | 0 | 0 | 16 | 16 |
| Total | 123 | 69 | 82 | 327 | 478 |

Table V shows the changes made to the `PowerAnalyzer` during refactoring. The `Timer` aspect required the most additions and the second most changes. The additions represent the lines that we added when we used the Extract Method Refactoring approach to refactor blocks of code as new functions so that these functions could begin with the `tmr` prefix. The changes represent the modifications we made to the code that called existing functions that were renamed.

The `Excepter` aspect required the most changes when code for checking and handling return values was removed, but it also resulted in the largest number of deleted lines of code from the `PowerAnalyzer`.

The lines added or modified in `PowerAnalyzer` for the `Excepter` aspect represent cases where we added local try/catch blocks to prevent applications from exiting on errors. The additions for the `CheckFwArgs` aspect are for converting the advised functions to static methods of classes. The `CadTrace` aspect does not replace any application code, but instead adds tracing information used to debug an application error. During normal use, it is disabled.

The `ViewCache` aspect required a change to one line of code and the removal of 21 lines, while `FetTypeChkr` and `FwErrs` each deleted fewer than twenty lines of code without requiring any change to the original application. The `UnitCvrt` aspect is not shown in Table V because the concern it represents was not implemented in the initial revision. Instead, the concern was introduced during the transition from revision two to revision three.

Table VI shows the changes made to the `ErcChecker` during refactoring. The `QueryPolicy` aspect enabled many deletions by refactoring common policy code for the `ErcQuery` class. The changes and additions required during refactoring varied among the different subclasses of `ErcPolicy` [18]. For 15 subclasses, we only made deletions. We added a method to 18 subclasses to consistently manage memory. The subclass consistency is necessary because the `QueryPolicy` aspect applies the same policy advice to all subclasses. In addition, six subclasses perform electrical checking in multiple phases, requiring that more than one location within the class be refactored.

Both the `QueryConfig` and `FwErrs` aspect enabled deleting about the same number of lines as the size of the aspect and required no changes and additions. The `CheckFwArgs` also removed approximately the same number

TABLE VI

ERCCHECKER: SOURCE CODE CHANGES MADE

| Aspect Name | Aspect Size(LOC) | Changes to primary code(LOC) | | | |
|---|---|---|---|---|---|
| | | Additions | Changes | Deletions | Total |
| QueryPolicy | 10 | 73 | 217 | 335 | 625 |
| QueryConfig | 14 | 0 | 0 | 57 | 57 |
| FwErrs | 16 | 0 | 0 | 24 | 24 |
| Caching | 130 | 0 | 0 | 184 | 184 |
| ErcTracing | 108 | 0 | 0 | 1057 | 1057 |
| CheckFwArgs | 29 | 0 | 3 | 33 | 36 |
| Excepter | 28 | 16 | 11 | 24 | 51 |
| Total | 335 | 89 | 231 | 1714 | 2034 |

of lines of code as the size of the aspect; the 3 lines added were to move functions into classes as static methods so that the aspect pointcut use the class name rather than a list of functions.

### C. Size

In the process of refactoring we discovered missing functionality that should have been implemented in each of the three applications. There were concerns, such as null pointer checking and tracing, which were not fully or consistently implemented in the original application. In `InstanceDrivers`, these concerns were related to the `Excepter`, `Tracing`, and `CheckFwArgs` aspects. In the `PowerAnalyzer`, these concerns were related to the `Excepter`, `CadTrace` and `CheckFwArgs` aspects. The concerns implemented inconsistently in `ErcChecker` were `CheckFwArgs`, `Excepter`, and `ErcTracing`.

Other than source code, we do not have access to documentation that communicates the rationale behind the coding decisions. Thus, we do not know whether or not some checking or tracing code was left out on purpose. However, there are several cases where the number of lines of concern code increased in later revisions of the applications. This indicated to us that the developers found a need to add the concern code to locations previously missed. Since one benefit of using aspects is to achieve consistency, a fair comparison of the size of the original and refactored applications would make it necessary to also compare the size of the equivalent application, which is the amount of code needed to achieve a consistent use of concern code in the applications.

In the equivalent implementation, the code would be added at each location advised by the aspect. Thus, we calculated the equivalent size by multiplying the number of join points advised by the aspect with the number of lines of code required in C++ to implement the advice. For example, code for a null pointer check would typically have 3 lines to check, print, and return. The size of the refactored application includes the refactored C++ code, the aspects, and any support code that is invoked from within the aspect advice.

The data for the six `InstanceDrivers` revisions is shown in Table VII. The ‘AOP’ column shows the size of the refactored application. The last column is the size of the refactored application (AOP) minus the size of

TABLE VII

INSTANCEDRIVERS: CODE SIZE (LOC)

| Revision | Original | Equivalent | AOP | AOP savings |
|---|---|---|---|---|
| 1 | 1619 | 1972 | 1924 | 48 (2.4%) |
| 2 | 1672 | 2025 | 1974 | 51 (2.5%) |
| 3 | 1823 | 2209 | 2130 | 79 (3.5%) |
| 4 | 2967 | 3473 | 3236 | 237 (6.7%) |
| 5 | 3008 | 3544 | 3269 | 269 (7.5%) |
| 6 | 3155 | 3743 | 3395 | 348 (9.3%) |

the equivalent application. This number represents the reduction in size achieved by using aspects and shows that this size reduction grew over time as the size of the application grew. Thus, even though using aspects did not initially reduce code size by much, these aspects required fewer code additions in later revisions resulting in a 9.3% reduction in size.

TABLE VIII

POWERANALYZER: CODE SIZE (LOC)

| Revision | Original | Equivalent | AOP | AOP savings |
|---|---|---|---|---|
| 1 | 13,931 | 14,066 | 13,734 | 332 (2.4%) |
| 2 | 14,861 | 15,000 | 14,638 | 362 (2.4%) |
| 3 | 15,183 | 15,322 | 14,959 | 363 (2.4%) |
| 4 | 15,356 | 15,499 | 15,128 | 371 (2.3%) |
| 5 | 15,727 | 15,876 | 15,482 | 394 (2.5%) |
| 6 | 16,412 | 16,567 | 16,118 | 449 (2.7%) |
| 7 | 16,600 | 16,755 | 16,294 | 461 (2.8%) |

Table VIII contains the data for the `PowerAnalyzer`.

New code was added at each revision for some crosscutting concerns, such as tracing. Thus, refactoring results in code savings with each revision because the changes are made only to the `Tracing` aspect. This increased savings in later revisions was particularly apparent for `InstanceDrivers` with the `Tracing` aspect.

In the `PowerAnalyzer`, 72% (332/461 lines) of the AOP savings were realized with the initial revision. The initial benefit was much larger for `PowerAnalyzer` 332 lines) than for `InstanceDrivers` (48 lines); refactoring saved more code in the initial revision than the size of the aspects themselves.

To implement the `CheckFwArgs` and `Tracing` aspects, we created a 200 LOC library that provides all the type-specific printing and checking methods needed to handle all the datatypes advised. The library can be accessed by any datatype-based aspect. Its development represents a one time cost for these aspects. The library increased the size of the `InstanceDrivers` application, but with greater functionality.

The data for three revisions of `ErcChecker` is shown in Table IX. The initial revision did not have checks.

There were three cases where a return code was not checked; we corrected this using the `Excepter` aspect. 320 methods in the classes used framework pointers without checking for null pointers; using `CheckFwArgs` fixed them. In revision four, four checks were missing in the original code; two were fixed by using the `Excepter` aspect and two were fixed by using the `CheckFwArgs` aspect.

TABLE IX

ERCCHECKER: CODE SIZE (LOC)

| Revision | Original | Equivalent | AOP | AOP savings |
|---|---|---|---|---|
| 1 | 51,692 | 52,664 | 50,492 | 2172 (4.1%) |
| 2 | 54,137 | 55,132 | 52,845 | 2287 (4.1%) |
| 3 | 64,145 | 65,154 | 62,608 | 2546 (3.9%) |

Like `PowerAnalyzer`, most of the AOP savings is realized in revision 1 of the `ErcChecker`. The ratio is even more pronounced for the `ErcChecker`, with 85% (2172/2546 lines) of the saving from revision 1. Refactoring removed more code for the `ErcChecker`, which was by far the largest application. In terms of percentage saved, `ErcChecker` had a large percentage saved than `PowerAnalyzer` but saved less than half of the percentage in the final revision of `InstanceDrivers`.

### *D. Performance*

We present the data for woven code size along with memory and execution time performance data in Tables X and XI. The first column lists the revision. The second column shows the multiplicative increase in the number of lines of the woven code. For example, 4.3x means that the woven code is 4.3 times larger than the original code. The third column shows the percent increase in compiled object code. The fourth column shows the percent increase in regression test execution time. The fifth column shows the percent increase in memory usage.

TABLE X

INSTANCEDRIVERS: REFACTORED APPLICATION PERFORMANCE DATA

| Revision | Code Increase | ObjCode Increase | Run Time Increase | Memory Increase |
|---|---|---|---|---|
| 1 | 4.3x | +54% | +5.1% | +2.4% |
| 2 | 4.3x | +62% | +5.1% | +2.3% |
| 3 | 4.4x | +91% | +5.0% | +2.1% |
| 4 | 3.9x | +119% | +14.0% | +11.1% |
| 5 | 4.0x | +120% | +15.0% | +12.1% |
| 6 | 4.2x | +123% | +18.0% | +15.0% |

Two factors explain the increases in `InstanceDrivers` memory usage and execution time for later revisions. First, more aspects were added to the application. Revision six has more aspect code woven in than revision one.

Second, the regression tests evolved with the application. In the fourth revision, tests were added that had tracing enabled by default unlike some of the previous tests.

Using any aspect increases woven code size since the aspect is implemented as a C++ class and each joinpoint has an associated class with joinpoint-specific data. This code bloat is a direct result of how AspectC++ implements aspect weaving. However, some aspects increase code size more than others, due to factors such as the number of joinpoints and features such as templates, which result in additional generated code at each joinpoint. For `InstanceDrivers`, `Tracing` and `CheckFwArgs` were responsible for 80% of the increase in woven code size.

TABLE XI

POWERANALYZER: REFACTORED APPLICATION PERFORMANCE DATA

| Revision | Code Increase | ObjCode Increase | Run Time Increase | Memory Increase |
|---|---|---|---|---|
| 1 | 1.78x | +30% | +10.0% | +14.0% |
| 2 | 1.79x | +29% | +9.3% | +12.6% |
| 3 | 1.80x | +29% | +9.1% | +12.1% |
| 4 | 1.79x | +29% | +9.1% | +12.0% |
| 5 | 1.79x | +29% | +9.0% | +12.0% |
| 6 | 1.80x | +29% | +9.0% | +12.0% |
| 7 | 1.80x | +29% | +9.0% | +12.0% |

The `PowerAnalyzer` data is shown in Table XI. The increases in `PowerAnalyzer` execution time, memory usage, and object size were slightly less with later revisions. A feature was added to the application in revision three that increased the run-time of each test by adding an extra processing step. Because this new feature was not advised by aspects, it increased the run time and memory usage for each batch-oriented test run without increasing the aspect-specific memory usage and execution time. Thus, the overhead from aspects was reduced during a given run.

The increase in the code size of the `PowerAnalyzer` was less than half the increase of `InstanceDrivers`. This difference was due to the greater use of aspects that employ template metaprogramming to recursively expand argument lists into individual typed values. Although template metaprogramming led to high code bloat in the `InstanceDrivers`, it was used in aspects (`Tracing` and `CheckFwArgs`) that are called infrequently during application regression tests, and have minimal impact on performance. The `PowerAnalyzer` aspects, such as `Excepter`, intercept calls that are made frequently during regression tests, which causes a higher performance overhead.

In the `PowerAnalyzer`, memory usage and execution times both consistently increased by about 10%, primarily due to the consistent error checking implemented in the `Excepter`, `CheckFwArgs`, `FetTypeChkr`, and `FwErrs` aspects. In addition, execution time also increased because the `CheckFwArgs` aspect processes each parameter; framework pointers are checked and non-framework parameters are ignored. Calling the empty virtual 'no-op' function for non-framework parameters also introduces some overhead. These checks provide improved

error detection and traceability without an order of magnitude increase in execution time and memory use, and reduce code size by eliminating scattered code to provide the same checks.

When we performed the original regression testing, test data for `ErcChecker` was no longer accessible to us for proprietary reasons. Thus, we only present woven code size and object code size data for the `ErcChecker`, which is shown in Table XII.

TABLE XII

ERCCHECKER: AOP WOVEN SIZE DATA

| Revision | Code Increase | ObjCode Increase |
|---|---|---|
| 1 | 4.8x | +125% |
| 2 | 4.7x | +123% |
| 3 | 5.1x | +136% |

The `ErcChecker`'s increases in woven code size and compiled object size are similar to `InstanceDrivers`. The `ErcChecker` makes extensive use of template-based aspects, as does, `InstanceDrivers`, which increases both woven code size and object code size.

We present join point data for `InstanceDrivers`, `PowerAnalyzer` and `ErcChecker` in Table XIII, using "n/a" for aspects that were not used in an application, and thus, are not applicable. This data is from the last revision of each application, thus all aspects were present in at least one of the applications. We see from the table that the `Excepter` was woven into more places in the `PowerAnalyzer` than the `InstanceDrivers` and `ErcChecker`. This explains why it had a greater impact on execution time and memory usage of the `PowerAnalyzer`.

The `Tracing` aspect advises many join points in `InstanceDrivers`, resulting in a large increase in the woven code size. Similarly, the `ErcTracing` aspect and `CheckFwArgs` aspects were woven in many join points of `ErcChecker`. Analysis of `ErcChecker` individual regression runs confirmed that when only the `Tracing` aspect was disabled, average memory and execution time increases were only between 2-4% rather than 9-10%. Since tracing is only enabled in rare cases for debugging problems, the average case for `InstanceDrivers` is a 2-4% increase. Since the verbose mode is only used for debugging, the negative performance impact in that case is acceptable because the aspect provides more thorough debug output.

### *E. Change Locality*

We measure change locality by comparing the number of modules and files that we modified when going from revision `N` to revision `N+1`. A lower number indicates better change locality. We also counted the number of lines changed to go from one version to the next.

Each row in Table XIV represents moving from one revision to the next one. For example, '1-2' in the 'Revisions' column means changes when moving from revision one to revision two. The 'Lines' and 'Lines(AOP)' columns show

TABLE XIII

ADVISED JOIN POINT DATA FOR ALL APPLICATIONS

| Aspect | Instance-Drivers Advised Join points | Power-Analyzer Advised Join points | ErcChecker Advised Join points |
|---|---|---|---|
| Caching | 3 | n/a | 27 |
| CadTrace | n/a | 8 | n/a |
| CheckFwArgs | 9 | 54 | 338 |
| ErcTracing | n/a | n/a | 211 |
| Excepter | 11 | 91 | 11 |
| FetTypeChkr | n/a | 14 | n/a |
| FwErrs | n/a | 9 | 12 |
| QueryConfig | n/a | n/a | 58 |
| QueryPolicy | n/a | n/a | 45 |
| Singleton | 4 | n/a | n/a |
| Timer | n/a | 34 | n/a |
| Tracing | 115 | n/a | n/a |
| UnitCvrt | n/a | 10 | n/a |
| ViewCache | n/a | 10 | n/a |

how many lines were modified in the original application and in the refactored application, respectively. Similarly, the 'Modules' and 'Modules(AOP)' columns compare how many modules were modified. Last, the 'Files' and 'Files(AOP)' columns compare how many source files were modified in the original and refactored application. The refactored value for each metric is shown next to the original for easy comparison. The table shows that better change locality (i.e. lower number of modules and files changed) generally corresponds with fewer lines changed.

TABLE XIV

INSTANCEDRIVERS: CHANGE LOCALITY

| Revisions | Lines | Lines (AOP) | Modules | Modules (AOP) | Files | Files (AOP) |
|---|---|---|---|---|---|---|
| 1-2 | 56 | 53 | 3 | 2 | 3 | 2 |
| 2-3 | 148 | 149 | 8 | 8 | 9 | 9 |
| 3-4 | 1155 | 1125 | 26 | 26 | 14 | 14 |
| 4-5 | 59 | 49 | 10 | 8 | 9 | 8 |
| 5-6 | 158 | 109 | 13 | 9 | 5 | 5 |

The increase of one line in the number of lines changed when going from revision two to revision three was caused by the implementation choice in the `CheckFwArgs` aspect: we refactored the functions needing their framework pointers to be checked into static class methods, and added an additional wrapper class to hide this change from the rest of the application. This increase was small, and had the original code used methods of a class rather

than functions, the AOP cost would be less. For all other revision changes, the refactored `InstanceDrivers` application had fewer line changes.

Aspects improved module change locality in three revisions and file change locality in two revisions. Module and file change locality were never worsened by refactoring. However, even when there were improvements, they tended to be small. This was because each new revision included changes in several core and crosscutting concerns at once. While aspects localized the changes in crosscutting concerns, the changes in the core concerns still had to be implemented in several modules and files.

The `PowerAnalyzer` change locality data is shown in Table XV. As in the `InstanceDrivers`, change locality either showed improvement or remained the same.

TABLE XV

POWERANALYZER: CHANGE LOCALITY

| Revisions | Lines | Lines (AOP) | Modules | Modules (AOP) | Files | Files (AOP) |
|---|---|---|---|---|---|---|
| 1-2 | 940 | 879 | 78 | 75 | 24 | 23 |
| 2-3 | 322 | 320 | 6 | 5 | 4 | 3 |
| 3-4 | 173 | 169 | 21 | 19 | 15 | 13 |
| 4-5 | 371 | 354 | 55 | 53 | 16 | 16 |
| 5-6 | 685 | 636 | 77 | 75 | 21 | 21 |
| 6-7 | 188 | 178 | 7 | 5 | 5 | 5 |

The `ErcChecker` change locality data is shown in Table XVI. As in the `InstanceDrivers` and `PowerAnalyzer` applications, refactoring provided a small decrease in module and file change locality. The small improvements were due to changes avoided because of the `ErcTracer` aspect. Like the `Timer` aspect in `InstanceDrivers`, the concerns refactored by `QueryPolicy` and QueryConfig had few changes during the evolution of the `ErcChecker`. Thus, these did not decrease module and file change locality.

TABLE XVI

ERCCHECKER: CHANGE LOCALITY

| Revs | Lines | Lines (AOP) | Modules | Modules (AOP) | Files | Files (AOP) |
|---|---|---|---|---|---|---|
| 1-2 | 4,638 | 4,546 | 112 | 109 | 47 | 46 |
| 2-3 | 16,967 | 16,722 | 317 | 310 | 105 | 103 |

### *F. Concern Diffusion*

In Table XVII we list each aspect used in `InstanceDrivers`, the number of concern switches removed by the aspect, and any new concern switches that occur when aspects are used. Thus, although we do not measure concern

diffusion over lines of code (CDLOC) directly, we measure the difference in CDLOC using the final revision of each application to manually compare the concern switch differences. The numbers in parentheses represent concern diffusion values when we consider equivalent functionality.

TABLE XVII

INSTANCEDRIVERS: CONCERN DIFFUSION

| Aspect | Concern Switch Reduction | New Concern Switches |
|---|---|---|
| Caching | 16 | 0 |
| CheckFwArgs | 0 (18) | 2 |
| Excepter | 16 | 0 |
| Singleton | 6 | 0 |
| Tracing | 78 | 0 |
| Total | 116(134) | 2 |

The `CheckFwArgs` aspect lists two values in the reduction column: 0 and 18. The value, 0, reflects that no checking was done in the original code, hence no concern switching actually occurred. The value, 18, represents the number of concern switches (2 switches per method in 9 methods) that would have occurred if the concern had been implemented in every place (equivalent functionality). The addition of two new concern switches reflects the restructuring that we performed in the core concern to allow us to use a simple pointcut as described in Section IV.

For the `InstanceDrivers` application, even without considering equivalent functionality, 116 concern switches were removed and only two were added.

TABLE XVIII

POWERANALYZER: CONCERN DIFFUSION

| Aspect | Concern Switch Reduction | New Concern Switches |
|---|---|---|
| CadTrace | 72 | 0 |
| CheckFwArgs | 84 | 8 |
| Excepter | 98 | 30 |
| FwErrs | 18 | 0 |
| FetTypeChkr | 28 | 0 |
| Timer | 68 | 0 |
| UnitCvrt | 38 | 0 |
| ViewCache | 20 | 0 |
| Total | 426 | 38 |

We show the reduction in concern diffusion of the final revision of `PowerAnalyzer` in Table XVIII. Overall, concern diffusion decreases in `PowerAnalyzer`. The `CadTrace` reduction reflects that the equivalent debugging activity would have resulted in 2 switches before and 2 switches after each of the 18 join points. The `Excepter` concern has a net decrease of 68 concern switches. At the locations where 30 concern switches were added, a

concern switch was also removed at the same location. Adding and removing a concern switch was done when refactoring error handling. In the original application, errors in functions (such as `fopen`) returned error codes that would be checked as follows:

```
fp = fopen(filename,mode);
if(fp==NULL) {
   //error handling code
}
//back to regular code
```

This shows two concern switches: (1) error-handling, for when the variable `fp` is NULL, and (2) the primary code concern for which the file pointer `fp` will be used.

For applications to catch errors locally, the developer must add a try/catch block around the function (`fopen`) rather than checking the return value, as shown below:

```
try {
   fp = fopen(filename,mode);
}
catch(anErrorOccurred e) {
   //error handling code
}
```

This represents two concern switches (switching to and from error handling code in the catch block). The net reduction in concern switches is 316 because Table XVIII shows 354 switches were reduced and 38 were added.

We show the reduction in concern diffusion of the final revision of `ErcChecker` in Table XIX.

TABLE XIX

ERCCHECKER: CONCERN DIFFUSION

| Aspect | Concern Switch Reduction | New Concern Switches |
|---|---|---|
| Caching | 216 | 0 |
| CheckFwArgs | 24 (38) | 0 |
| Excepter | 14 (668) | 8 |
| ErcTracing | 598 | 0 |
| FwErrs | 12 | 0 |
| QueryConfig | 114 | 0 |
| QueryPolicy | 78 | 57 |
| Total | 1,056 (1,724) | 65 |

The `CheckFwArgs` and `Excepter` aspects list two values in the reduction column. The smaller value reflects

the concern switches reduced in the original code. The larger parenthesized value represents the number of concern switches that would have occurred if the concern had been implemented in every place (equivalent functionality).

Each aspect reduced the number of concern switches. The `QueryPolicy` aspect required adding a new virtual method to the `ErcQuery` base class, and 14 of the 58 classes had to override this method. The changes to the base class and 14 sub-classes are where the 57 new concern switches were added. The `Excepter` class reduced 14 switches, but 8 new concern switches were added as local try/catch blocks to handle some errors locally rather than exiting from the application. All other aspects reduced concern switches without adding new concern switches. A total of 1,020 concern switches were reduced in the original application.

### *G. Test Coverage*

We describe the challenges and trends we observed when we gathered test coverage data for four revisions of `InstanceDrivers` and PowerAnalyzer. As previously mentioned, regression data was not available for `ErcChecker`.

Because AspectC++ generates source code when weaving, tools such as `gcov` instrument the woven source code in order to gather coverage data. Thus, output from `gcov`, such as source code annotated with the number of times each line was executed, is based on the woven code. In addition, coverage data is based on the woven code rather than on the source code the developer edits. Since the woven code contains new code generated through aspect weaving and an extra level of indirection for the advice to intercept and advise methods and functions, it made the report of missed lines more difficult to understand.

In Table XX we show statement coverage data for four revisions of `InstanceDrivers`. The first column lists the revision, while the second and third column show covered lines of code and total lines for the original application, and the fifth and sixth columns show covered and total lines of woven code of the refactoring application. The fourth and seventh columns show the percentage of covered lines of code for the original and refactored applications. For this study, we used the existing regression tests as is; we did not change them to try and achieve 100% coverage in the original or refactored applications.

TABLE XX

INSTANCEDRIVERS: STATEMENT COVERAGE

| Revision | Covered lines | LOC | Percentage | Covered lines(AOP) | LOC (AOP) | Percentage (AOP) |
|---|---|---|---|---|---|---|
| 1 | 1517 | 1619 | 94% | 6458 | 7082 | 91.2% |
| 3 | 1651 | 1767 | 93% | 7355 | 8040 | 91.4% |
| 4 | 2673 | 2967 | 90% | 11150 | 11578 | 96% |
| 6 | 2835 | 3155 | 90% | 12602 | 13140 | 96% |

From Table XX, we observe that the refactored application always had more lines of missed code. Although achieving 100% coverage is not feasible in practice, developers may wish to check which lines were missed. Thus,

additional missed lines will increase the time spent manually evaluating test results. The missed lines correspond to the primary code that did not get executed as well as the aspect code. If statements in the primary code are not executed by the test inputs, then any associated join point-specific code will also be missed. Moreover, the woven code can also contain unreachable statements. Each join point has methods, such as `Joinpoint::signature()`, that can be used in the advice body of the aspect. If these join point methods are not used by any advice, they are unreachable and will be marked as not covered by statement coverage tools. The developer can assume that the weaver produces correct code, so these missed lines may not always be a cause for concern.

Table XX reflects two significant changes that occurred in revision four of `InstanceDrivers`. The first change was that more than 1000 lines of code were added, the largest one-time increase for any revision. Secondly, because of the changes in the original code, the regression tests were updated by the original developers to test additional functionality, including more testing of the verbose-mode functionality, which provided much better testing for the `Tracing` aspect. Although the number of lines missed in revision four was still greater in the refactored version, the refactored version had a lower percentage of misses because previously missed `Tracing` aspect code was now being tested.

TABLE XXI

POWERANALYZER: STATEMENT COVERAGE

| Revision | Covered lines | LOC | Percentage | Covered (AOP) | LOC lines (AOP) | Percentage (AOP) |
|---|---|---|---|---|---|---|
| 1 | 11,496 | 13,931 | 82.6% | 21,244 | 24,744 | 85.9% |
| 3 | 12,591 | 15,183 | 82.9% | 23,481 | 27,292 | 86.0% |
| 5 | 12,937 | 15,727 | 82.3% | 24,156 | 28,151 | 85.8% |
| 7 | 13,395 | 16,600 | 80.7% | 25,293 | 29,760 | 85.0% |

Test data for four revisions of `PowerAnalyzer` is shown in Table XXI. The missing regression test cases lower the coverage. Because our focus was on the refactoring, attempting to obtain the old test cases to construct new test cases was beyond the scope of this study.

TABLE XXII

INSTANCEDRIVERS: JOIN POINT COVERAGE

| Revision | Covered Join points | Total Join points | Percentage |
|---|---|---|---|
| 1 | 54 | 77 | 70% |
| 3 | 54 | 84 | 64% |
| 4 | 85 | 114 | 75% |
| 6 | 107 | 145 | 74% |

Table XXII shows join point coverage for the same revisions for which we reported statement coverage. Because we have refactored the application to use aspects, we use join point coverage to measure what percentage of advised

join points executed by existing tests. Although we did not change or add tests for these studies, in practice the missed join points could be used to add tests for more thorough integration testing of aspects.

The first column contains the revision number, the second column shows the number of covered (executed) join points, while the third column shows the total number of join points. The last column (Percentage) is the percentage of join points covered. At revision three, the join point coverage decreased because there were new join points in the application which were not executed by the regression tests, even though the old join points were covered. At revision four, when the regression tests were improved so that the `Tracing` aspect was executed more often, we see an increase in join point coverage.

TABLE XXIII

POWERANALYZER: JOIN POINT COVERAGE

| Revision | Covered Join points | Total Join points | Percentage |
|---|---|---|---|
| 1 | 93 | 155 | 60% |
| 3 | 113 | 183 | 62% |
| 5 | 122 | 192 | 64% |
| 7 | 134 | 211 | 64% |

Table XXIII shows join point coverage for the `PowerAnalyzer`. Unlike `InstanceDrivers`, no changes occurred in the regression test suite of `PowerAnalyzer` between revisions. Thus, join point coverage percentage for `PowerAnalyzer` remained about the same across all revisions.

### *H. Defect Tracking*

We are interested in understanding what types of defects might be avoided with aspects, as well as what types of defects are introduced when using aspects or refactoring a application. For each revision, Table XXIV shows the number of defects identified in the original `InstanceDrivers` during refactoring and the number of defects that were introduced (and fixed) when refactoring the application.

In the original `InstanceDrivers` application, the two defects found in revision one were both failures to check the return code of `getenv()` calls for errors. These were fixed by using the `Excepter` aspect. One defect in revision four was also related to not checking `getenv()` error codes. The second defect was that the caching mechanism, which was manually implemented, did not cache the result in one case. This defect lowered performance but did not give incorrect results. This was fixed when we used our `Caching` aspect.

The first defect that we introduced in revision one was caused by the `Tracing` aspect, which accessed a class member variable when it advised a static method resulting in a null pointer access. The defect was fixed by making sure that we checked the '`this`' pointer of the advised object in the aspect so that member data was only inspected by non-static methods.

TABLE XXIV

INSTANCEDRIVERS: DEFECTS

| Revision | Defects Identified in Original Application During Refactoring | Defects Introduced When Refactoring Application |
|---|---|---|
| 1 | 2 | 2 |
| 2 | 0 | 0 |
| 3 | 0 | 1 |
| 4 | 2 | 0 |
| 5 | 0 | 0 |
| 6 | 0 | 0 |

The second defect we introduced in revision one was that the template-based C++ support code for checking methods with an arbitrary number of parameters did not work with zero-argument methods. We tested the aspect with a variety of parameter types using a small mock system [21], but this testing did not include a zero argument method.

At revision three, the woven code would not compile. The reason was that the `CheckFwArgs` aspect called a method that had to be defined and overridden for each framework pointer type used. The use of a new framework pointer as an argument caused a compilation error; we corrected this by defining the method for the newly used type.

TABLE XXV

POWERANALYZER: DEFECTS

| Revision | Defects Identified in Original Application During Refactoring | Defects Introduced When Refactoring Application |
|---|---|---|
| 1 | 22 | 2 |
| 2 | 1 | 0 |
| 3 | 0 | 0 |
| 4 | 2 | 0 |
| 5 | 2 | 0 |
| 6 | 2 | 1 |
| 7 | 0 | 0 |

We show the number of `PowerAnalyzer` defects in Table XXV. Of the 22 original defects in revision one, one defect was the invalid framework method call detected using the `CadTrace` aspect. 12 defects resulted from failure to check return values, and nine resulted from failure to check for null framework parameters. We removed these defects when we used the `Excepter` and `CheckFwArgs`. In revisions two and four, the original code contained defects due to missed unit conversions, which were fixed with the `UnitCvrt` aspect. Also, in revision four the

original code had a defect that was fixed by using the `Timer` aspect. Revisions five and six each contained two defects from failing to check for null framework parameters.

We introduced two defects when we refactored revision one. One defect was due to an incorrect pointer reference, while the other was due to incorrect try/catch logic added. We introduced a defect in revision six where one of the methods that was supposed to be advised by the `Excepter` aspect was not matched by the pointcut.

TABLE XXVI

ERCCHECKER: DEFECTS

| Revision | Defects Identified in Original Application During Refactoring | Defects Introduced When Refactoring Application |
|---|---|---|
| 1 | 325 | 2 |
| 2 | 8 | 1 |
| 3 | 5 | 1 |

We removed two defects from the `ErcChecker` when refactoring revision one: an `ErcQuery` did not write its results to a log file, and an `ErcQuery` could not be disabled with run-time configuration commands. We fixed the first defect by using the `QueryPolicy` aspect and fixed the second using the `QueryConfig` aspect. We identified three function calls whose return value was not checked; using the `Excepter` aspect provides these checks in the refactored version. We found 320 methods that used framework pointers without checking for null, which we corrected by using the `CheckFwArgs` aspect.

We introduced two aspect-related defects in revision one. In both cases, we failed to remove code from the application that was replaced by the aspect's advice. This caused both the aspect and the remaining code to try to free memory.

In revision two, refactoring removed eight defects from the original application. The body of an `ErcQuery` class method missed the deletion of a query object from memory. We removed this defect by using the `QueryPolicy` aspect. We identified seven missing null pointer checks in revision two. The `Excepter` aspect handled two of them and `CheckFwArgs` handled five. We encountered one refactoring error in revision two where new functions did not match the `ErcTracing` aspect's pointcut, disabling tracing for those functions.

We identified five defects in revision three of the original application. One defect resulted from failure to check for framework initialization errors; using the `FwErrs` aspect provided this check. We found two defects where the checking provided by the `CheckFwArgs` aspect was missing, and two defects where the checking performed by the `Excepter` aspect was missing. We encountered one refactoring error in revision three: the `Excepter` aspect did not advise a method that we expected it to advise. We detected this omission using our own weave analysis tool because the number of join points advised did not change as expected.

Eight of the nine refactoring defects found in the `PowerAnalyzer`, `InstanceDrivers`, and `ErcChecker` can be categorized into three fault types identified in our prior work [17]:

- Incorrect advice behavior: two defects from revision one of `InstanceDrivers`; one defect in revision two of `ErcChecker`.
- Advice syntax errors: one defect from revision three of `InstanceDrivers`.
- Pointcut too weak: one defect in revision six of `PowerAnalyzer`; one defect in revision three of `ErcChecker`.
- Accidental code duplication: two defects found in revision one of `ErcChecker`.

We also encountered a fault type that did not occur in our prior work [17]: faults introduced in the core concern during refactoring (two defects in revision one of `PowerAnalyzer`). These faults are not due to 'Accidental code duplication' because they involve errors introduced when using aspects, rather than failing to reduce code replaced by aspects. We observed that refactoring removed or avoided more defects than were introduced, and our regression tests identified the refactoring defects listed above.

## VI. Discussion

In this section we provide an overall analysis of the common results and observations from our study of these three applications. We also list threats to validity that may limit the applicability or affect the results.

### *A. Analysis of the results*

For all three applications, using aspects requires a time investment to develop aspects and refactor the original application. However, more than half of the refactoring changes were deletions, and the overall effect was to reduce source code size. There were increases in execution time, memory requirements, and compiled object size. The largest increase was for object size, which indicates potential memory and execution time increases. However, for these applications, object code size itself was not critical, since large increases in object code size resulted in small increases in memory requirements and execution time. Memory and execution time increases were caused by the safety checks in the refactored version that the original application omits. Had the original application implemented the same checks, the increases in execution time and memory usage would have been smaller. In addition, aspects that perform these checks can easily be modified by a change in the aspect, while modifying the equivalent checking code in the original application will require many changes. Refactoring, if not done carefully, can introduce faults.

Change locality was improved by using aspects. In general, refactoring makes it possible to avoid many changes in the core concerns. However, in our study, we found that refactoring caused two kinds of changes to the aspects or support code. First, for aspects such as the `Excepter`, whose pointcuts consisted of a list of functions, we needed to update the pointcut in subsequent revisions with additional function names to advise the new functions. Aspects, such as `CheckFwArgs`, whose pointcut was based on application code being part of a class or namespace, require that the core concerns adhere to that naming convention. Thus, new functions added to the core concern, which needed to be checked by the `CheckFwArgs` aspect, had to be enclosed within a class or namespace so that the pointcut matched them.

Concern diffusion was reduced as aspects provided better modularity than the original object-oriented implementation. The use of aspects removed defects that occurred when a concern was inconsistently implemented in multiple locations.

The benefits from using aspects in the three applications varied. For example, the `ErcChecker` had the largest reduction in source code size when revision one was refactored. By contrast, `InstanceDrivers` had little initial code savings, but later revisions saved more code because several crosscutting changes were avoided in the aspect-oriented implementation. Both the `ErcChecker` and the `PowerAnalyzer` realized more than half of the code savings during refactoring of revision one. The potential benefit of refactoring depends on the amount of crosscutting code in the original application. The final revision of `PowerAnalyzer` had 16,286 lines of code, with refactoring providing a savings of 397 (2%). The final revision of the `ErcChecker` contained 62,608 lines of code, with refactoring providing a savings of 2546 lines (4.1%). The final revision of `InstanceDrivers` had the highest percentage of crosscutting code savings, with 348 lines saved out of 3395 (10%).

In general, aspects that allow deleting the most lines of code do so by advising many join points. For team-based development, one potential challenge may be in communicating these relationships between the advice and the join points. As Griswold et al. [30] report, the obliviousness that name-based pointcuts provide may complicate maintenance and parallel development of aspects and core concerns. For our study, since one developer performed the refactoring and maintenance between revisions, this was not a challenge, but it is a potential challenge when refactoring legacy applications that are being maintained by a large team.

In our study, the aspects themselves changed little or not at all. For example, the `TimeEvent` class used by the `Timer` aspect of `InstanceDrivers` did not change. Had the type of timing data or core concerns that collected this data changed, the `Timer` aspect itself would have required changes. However, our study did not encounter this type of change.

### *B. Threats to Validity*

Like most case studies, it is difficult to generalize from a study of three applications. Thus, there are threats to external validity. This study applied the refactoring process to only three legacy applications. These applications were not selected randomly, which limits external validity. In addition, the applications were from the same problem domain. Choosing related applications that use a common framework may bias the applications toward certain design and coding styles and characteristics. Moreover, different results are likely when applying the process to applications in other domains.

Because of the limited nature of the evaluation, there are threats to internal validity – whether aspect-orientation is actually responsible for the improvements in maintainability.

One concern is that the same subject developed the aspects and refactored the three applications. Moreover, as part of his job, this subject had developed one application and performed maintenance on the other. The analysis of changes between revisions could have been influenced by the prior experience with the applications. When identifying aspects from the first revision of a legacy application, a developer may be biased toward parts of an

application that are known to be more change-prone, and may be biased against those parts of the application that were not change prone. A different subject and might choose different aspects and implement them differently. The approach in this study was to use all the aspects that were identified. Moreover, we only made changes that were necessary for using aspects. Another approach would be to favor more extensive refactoring (e.g., adding more object-orientation to the primary code) before using aspects.

The developer who created the aspects reported the defects avoided by aspects and the defects caused by aspects. A different developer might find different defects in the original application and insert different defects during refactoring. In addition, any aspect developer might be unaware of defects in his or her refactored application that would be found by another developer or if the test coverage were higher.

Using legacy applications to perform the study may pose problems. Legacy software often reflects the decisions made based on the language used and tools available at the time of developing the software. Legacy code may be dated by a particular design style, such as the use or lack of design patterns, which may affect what types of refactoring can be performed. Some changes, such as those related to a concern that crosscuts many files, may not have been made in the original code because of the cost; however, had the original implementation used aspects from the beginning, such changes would have been less costly.

How we use source code repositories of legacy applications presents an additional threat to internal validity. The revisions we considered are major releases across a large set of files. In between two major releases (coarse-grained revisions) of an application, two files might have different revision histories – one might not have changed at all, while the other might have had several file revisions. For files that have multiple changes between revisions, a module in a file might have undergone several changes, only one of which is related to a crosscutting concern. When that concern is refactored as an aspect, there will be a reduction in the amount of change in that module. However, since there are other changes in the module and the associated file, refactoring alone is unable to reduce file and module change locality. A more fine-grained approach would consider each file's individual revision history. Using fine-grained revisions for each file is likely to affect change locality results because each change would be smaller, and some changes would be related only to crosscutting concerns.

Construct validity focuses on whether the measures used represent the intent of the study. We compared the original and refactored applications over the existing revision history, using code-based metrics such as lines of code, number of modules and files modified, and code concern diffusion. This is one approach for studying the effects of aspect-oriented technology on maintainability. Other researchers have created two separate applications [10], [30], one designed with aspects and one designed without aspects, rather than modifying the original application to incorporate aspects. Maintenance was simulated by implementing features defined in use cases. We identified crosscutting concerns from source code and did not use requirements or design documents. A different approach would be to define aspects from design documents or indications of designer intent.

Our results are dependent on the features and capabilities of AspectC++, which depend on the features of C++. The only effect that was a direct result of using AspectC++ was the code bloat resulting from templates in C++ and the relative lack of maturity of the AspectC++ implementation. AspectC++ is based on the design and goals

of AspectJ [34], but differences between C++ and Java are reflected in AspectC++. For example, as C++ lacks reflection, access to parameter type information is provided through a static, compile-time API in AspectC++. Language differences between C++ and Java, such as pointers, memory management, and operator overloading affect how design patterns such as singleton can be implemented, and cause AspectC++ not to support some features, such as getter and setter accesses of class attributes. Moreover runtime weaving is possible in AspectJ but not in AspectC++, which may impact performance results. Studies using a different primary code language and a different aspect-oriented language may encounter different challenges and benefits from refactoring.

## VII. Related Work

Coady and Kiczales [5] reported finding benefits to aspect-oriented refactoring, including more localized change and reduced redundancy of scattered concern code. Our use of the source code repository of an existing application is similar to theirs, but we consider three applications rather than one, and we follow the changes across more revisions. We used similar measures, although we did not include directory change locality. Coady and Kiczales measured the number of source code locations (i.e. blocks of code) that change, which would correspond to changes in concern diffusion that we measured. In addition, we do not restrict the focus to the set of aspects identified in the first revision, but also look for additional aspects in subsequent revisions.

Our study differs from Kulesza et al. [10] by using legacy applications as candidates for refactoring, and also by evaluating the effects over many revisions rather than two. In addition, the revision differences we consider were based on actual changes, which may differ from the changes that are related to a design change. Since we use legacy applications, our results also indicate what types of aspects occur in real applications, which avoids the bias of creating an application after having been exposed to aspect-oriented programming. Kulesza et al. reported a 12% reduction in code when using aspects, which is slightly higher than the `InstanceDrivers` results. They also found that aspects improve maintainability by improving cohesion and by reducing the number of changes. Our results on industrial-scale applications agree with their findings that the benefits of AOP scale with program size.

Bruntink et al. [7] refactored a crosscutting concern that implements the return code idiom in a large (15 million LOC) application. However, their focus is on the variability of this idiom and the challenges that occur during refactoring. We reported similar variability challenges and results in previous work [20] while exploring different implementation approaches for the `Excepter` aspect. Rather than focusing on a single concern in one application, in this paper we focus on multiple aspects across multiple revisions of legacy applications.

Ceccato and Tonella [13] evaluate the relationship between their cohesion and coupling metrics, and maintenance and understanding effort using two implementations of the observer pattern while we use three legacy applications over multiple revisions. They do not demonstrate that the measures satisfy properties of cohesion and coupling [35].

Binkley et al. [36] developed the AOP-Migrator, an Eclipse plugin that implements six refactorings from OOP to AOP. The tool requires the user to be deeply involved in the approach for extracting code. A study was performed to refactor four applications ranging from 11.6KLOC to 40KLOC, without considering later revisions. They searched for instances of the six refactorings in the code. We attempted to find all the crosscutting concerns we could

identify. They counted various *contains* and *use* dependencies: base-to-base, base-to-concern, concern-to-concern, and concern-to-base and report a reduction in the concern-to-base dependencies. Their base code is completely oblivious of the concern code. The reported reduction in base code size is limited. The execution times showed no specific tendency; sometimes the original was faster and sometimes the refactored version was faster.

Griswold et al. [30] refactored HyperCast, a large Java application for multicast networks, using AspectJ. They encountered difficulty in specifying pointcuts that would match the state-machine of HyperCast because they needed to specify multiple points within a method. They reported that many pointcuts were too tightly coupled to names, so that changes to primary classes would break them. They reported two types of development problems. First, the tight coupling between aspects and method names prevented the development of aspects in parallel with primary code refactoring because the aspects could only be developed after inspecting the core concerns. Second, they encountered cases where join points were not accessible because the language (AspectJ) supports specifying join points at the method call level and data member level, but not at the `if` or `switch` statement level [37]. They propose the use of crosscutting interfaces to improve the modularity of aspect-oriented applications, avoid fragile pointcuts, and enable parallel development of aspects and classes. In our study, we made some changes to the core concerns to enable less fragile pointcuts, including renaming functions to begin with `tmr` and refactoring a collection of procedural functions as static methods of a class. These changes allowed us to use the existing pointcut mechanisms of AspectC++. We did not encounter problems relating to parallel development of aspects and core concerns because in our studies the core concerns already existed and the aspects being developed were based on the pre-existing core concerns. We did not apply their crosscutting interface approach, although it could be applied to a study like ours.

Figueiredo et al. [12] consider multiple revisions of two software product lines and report that using aspects had positive impact. However, in the context of product lines, they reported that aspects provided better design stability only for optional or alternative features, and did not perform as well for required features. Although our applications have complex features that may only apply to certain circuit styles or to certain electrical views, the entire application is always present with all features at each revision. In addition, our metrics focus not just on change-proneness but also on the benefits of code reduction and on the costs of development, performance overhead, and impacts to testability.

Bartsch and Harrison [11] created two online shopping applications, and had groups of subjects perform maintenance tasks on one of two separate applications, which were intended to be equally difficult to modify. Subjects performed tasks on one of the applications and completed a survey, which included information about how long certain tasks took. Bartsch and Harrison reported that the results appeared to slightly favor the object-oriented approach over the aspect-oriented approach. Although the test subjects were software professionals, the application was not an industrial application and the same tasks were evaluated repeatedly, rather than evaluating maintenance over multiple revisions.

Tsang, Clarke, and Baniassad [15] extended the Chidamber and Kemerer metrics and compared two real-time applications, one created using real-time Java extensions and the other created with Java and AspectJ. They did not

refactor an application, but instead compared metrics of two applications to determine strengths and weaknesses of aspect-oriented programming. They found that aspects improved modularity by reducing coupling and cohesion. However, aspects increased metrics such as weighted metrics per class, since a method in an object-oriented application often corresponded to a method plus associated advice in the aspect-oriented application. Our study uses metrics to compare three applications, but we compared a refactored application to its original. We focus on maintainability metrics such as size, change-proneness, and concern diffusion.

Hoffman and Eugster [16] consider multiple applications but they did not focus on maintainability over multiple revisions. Instead, they compared Java, AspectJ, and a new language mechanism they proposed, explicit join points, in order to provide reusable aspects that were modular without the problems found with obliviousness. Their study focuses on evaluating and improving aspect-oriented languages, while we focus on how refactoring can improve maintainability.

## VIII. Conclusions

Aspect-oriented refactoring of legacy applications promises to improve understandability and maintainability by replacing crosscutting code with aspects. The benefits of performing aspect-oriented refactoring of legacy applications need to be balanced against the costs of performing the refactoring.

This study examined the costs and benefits of refactoring three legacy applications in one application domain — VLSI CAD software — developed by one company — Hewlett-Packard. A total of 14 aspects were created; four of these aspects were used in more than one of the applications.

The costs of refactoring these systems include the human effort required to perform the refactoring, added code that will need to be maintained, faults introduced during the process, and negative effects on system performance:

- Effort: It took a total of ten hours for one developer to create and test the 14 aspects. The median time required was 35 minutes. One aspect required only 15 minutes to create and test; the most difficult aspect required 65 minutes.
- New aspect code created: Aspects ranged in size from 4 to 130 lines of code with a median aspect size of 15 lines. The mock systems used to test the aspects ranged from 15 to 600 lines of code, with a median size of 35 lines.
- Faults introduced: Refactoring introduced 3 or 4 defects in each system. These defects were identified through regression testing.
- Performance degradation: Depending on the system and revision, refactoring increased execution time from 5.1% to 18%, and increased memory requirements from 2.4% to 15%.

The most serious cost of refactoring these systems is the degraded performance, which might be mitigated with improved compiler and run-time aspect support. Other than system performance, the costs of refactoring were incurred when refactoring the first version.

The benefits of refactoring these systems include a net reduction in source code that must be maintained, simplification of maintenance activities in terms of fewer modules and files that must be adapted between versions

and reduction in concern diffusion, and identification of faults in the original applications:

- Reduction in code: Depending on the system and revision, refactoring saved from 2.4% to 9.3% of the total volume of source code.
- Fewer modules and files changed during maintenance: The modifications needed to evolve these systems required changes to fewer modules and fewer files in the refactored systems when compared to the original. The reduction of the number of modules and files changed between revisions was as high as 33%, with a median reduction of 6.7% in the number of modules changed and 3.1% in the number of files changed.
- Reduction in concern diffusion: Refactoring the three systems provided a net reduction of 114, 388, and 991 concern switches.
- Faults found through refactoring: We found 4, 29, and 338 program faults in the original applications during the refactoring process.

These results are from the refactoring of only three systems in one application domain. Different results might be obtained on different systems and in different domains. However, this study demonstrates that aspect-oriented refactoring can reduce code volume and the number of modules and files that need to be modified during maintenance. The refactoring activity also identified several program faults. The effort required to create and test the aspects was fairly modest — only ten hours. A reduction in performance was of greatest concern. This problem might be reduced through improvements in aspect-oriented technology. In particular, we need improvements in aspect support for C++.

This work shows that aspect-oriented refactoring can simplify maintenance by limiting the number of software items that must be revised, and can reduce program faults. We still need to know if the benefits will continue over the longer term and will occur for software in other application domains.

## REFERENCES


[1] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, ser. Lecture Notes in Computer Science, M. Akşit and S. Matsuoka, Eds. New York, NY: Springer-Verlag, June 1997, vol. 1241, pp. 220–242.

[2] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[3] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, ser. ACM SIGPLAN Notices, C. Norris and J. J. B. Fenwick, Eds., vol. 37, 11. New York: ACM Press, Nov. 4–8 2002, pp. 161–173.

[4] C. Zhang and H.-A. Jacobsen, *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 11, pp. 1058–1073, November 2003.

[5] Y. Coady and G. Kiczales, "Back to the future: A retroactive study of aspect evolution in operating system code," in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD-2003)*, M. Akşit, Ed. ACM Press, Mar. 2003, pp. 50–59.

[6] J. P. Zagal, R. S. Ahus, and M. N. Voehl, "Maintenance-oriented design and development: A case study," *IEEE Software*, vol. 19, no. 4, pp. 100–106, 2002.

[7] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé, "Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations," in *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2007, pp. 199–211.

[8] V. R. Basili and D. M. Weiss, ""A Methodology for Collecting Valid Software Engineering Data"," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 6, pp. 728–738, November 1984.

[9] C. Koppen and M. Störzer, "PCDiff: Attacking the fragile pointcut problem," in *European Interactive Workshop on Aspects in Software (EIWAS)*, K. Gybels, S. Hanenberg, S. Herrmann, and J. Wloka, Eds., Sept. 2004. [Online]. Available: http://www.topprax.de/EIWAS04/EIWAS-Papers.zip

[10] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena, "Quantifying the effects of aspect-oriented programming: A maintenance study," in *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 223–233.

[11] M. Bartsch and R. Harrison, "An exploratory study of the effect of aspect-oriented programming on maintainability," *Software Quality Journal*, vol. 16, no. 1, pp. 23–44, 2008.

[12] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, and F. Dantas, "Evolving software product lines with aspects: an empirical study on design stability," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 261–270.

[13] P. Tonella and M. Ceccato, "Refactoring the aspectizable interfaces: An empirical assessment," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 819–832, 2005.

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

[15] S. L. Tsang, S. Clarke, and E. Baniassad, "An evaluation of aspect-oriented programming for Java-based real-time systems development," in *Proceedings of The 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2004, pp. 291–300.

[16] K. Hoffman and P. Eugster, "Towards reusable components with aspects: an empirical study on modularity and obliviousness," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 91–100.

[17] M. Mortensen, S. Ghosh, and J. Bieman, "Testing during refactoring: Adding aspects to legacy systems," in *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE 06)*, Raleigh, North Carolina, USA, November 2006.

[18] M. Mortensen and S. Ghosh, "Using aspects with object-oriented frameworks," in *AOSD '06: 5th International Conference on Aspect-oriented Software Development Industry Track*, Bonn, Germany, March 2006, pp. 9–17.

[19] ——, "Creating pluggable and reusable non-functional aspects in AspectC++," in *ACP4IS '06: Proceedings of the 5th workshop on Aspects, components, and patterns for infrastructure software*, Bonn, Germany, 2006.

[20] ——, "Refactoring idiomatic exception handling in C++: Throwing and catching exceptions with aspects," in *AOSD '07: 6th International Conference on Aspect-oriented Software Development Industry Track*, Vancouver, British Columbia, Canada, March 2007, pp. 9–15.

[21] M. Mortensen, S. Ghosh, and J. Bieman, "A test driven approach for aspectualizing legacy software using mock systems," in *Information and Software Technology*, vol. 50, no. 7-8. Elsevier, 2008, pp. 621–640.

[22] R. T. Alexander, J. M. Bieman, and A. A. Andrews, "Towards the Systematic Testing of Aspect-Oriented Programs," *Technical Report CS-4-105, Department of Computer Science, Colorado State University*, March 2004.

[23] C. Clifton and G. T. Leavens, "Obliviousness, modular reasoning, and the behavioral subtyping analogy," in *SPLAT 2003: Software engineering Properties of Languages for Aspect Technologies at AOSD 2003*, Mar. 2003, available as Computer Science Technical Report TR03-01a from ftp//:ftp.cs.iastate.edu/pub/techreports/TR03-01/TR.pdf.

[24] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.

[25] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and Schrder-Preikschat, "A quantitative analysis of aspects in the eCos kernel," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 191–204, 2006.

[26] J. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander, "Design patterns and change proneness: An examination of five evolving systems," in *Proceedings of the $9^{th}$ international Software Metrics Symposium*, M. Berry and W. Harrison, Eds. IEEE Computer Society Press, September 2003, pp. 40–49. [Online]. Available: csdl.computer.org/comp/proceedings/metrics/2003/1987/00/19870040abs.htm

[27] M. Bruntink, A. van Deursen, and T. Tourwe, "Isolating idiomatic crosscutting concerns," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 37–46.

[28] D. Lohmann, O. Spinczyk, and W. Schrder-Preikschat, "On the configuration of non-functional properties in operating system product lines," in *ACP4IS: Aspects, Components, and Patterns for Infrastructure Software*, D. H. Lorenz and Y. Coady, Eds., Mar. 2005.

[29] L. C. Briand, W. J. Dzidek, and Y. Labiche, "Instrumenting contracts with aspect-oriented programming to increase observability and

support debugging," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 687–690.

[30] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan, "Modular software design with crosscutting interfaces," *IEEE Software*, vol. 23, no. 1, pp. 51–60, 2006.

[31] D. Wampler, "Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces," 2006, pp. 27–30.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley, 1995.

[33] O. Spinczyk, D. Lohmann, and M. Urban, "AspectC++: An AOP extension for C++," in *Software Developers Journal*, no. 7. Software-Sydawnicto, Warsaw, Poland, June 2005, pp. 68–74.

[34] O. Spinczyk, A. Gal, and W. Schrder-Preikschat, "AspectC++: an aspect-oriented extension to the C++ programming language," in *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2002, pp. 53–60.

[35] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, 1996.

[36] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, "Tool-supported refactoring of existing object-oriented code into aspects," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 698–717, 2006.

[37] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, "Information hiding interfaces for aspect-oriented design," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2005, pp. 166–175.