

# Using Aspects with Object-Oriented Frameworks

Michael Mortensen  
Hewlett-Packard  
3404 E. Harmony Road, MS 88  
Fort Collins, CO 80528  
and  
Computer Science Department  
Colorado State University  
Fort Collins, CO 80523  
mmo@fc.hp.com

Sudipto Ghosh  
Computer Science Department  
Colorado State University  
Fort Collins, CO 80523  
ghosh@cs.colostate.edu

## ABSTRACT

We investigate potential uses of aspect-oriented programming in the context of object-oriented C++ frameworks used in the development of VLSI CAD applications. We use existing applications to explore the use of different kinds of aspects. We differentiate between framework-based aspects and application-specific aspects. Framework-based aspects modularize cross-cutting code based on how an application uses or extends an object-oriented framework. We propose the use of a library of framework-based aspects that can be developed for and leveraged across a family of framework-based applications. Application-specific aspects allow modularizing existing cross-cutting code in VLSI CAD applications. Preliminary results for each type of aspect are presented, along with challenges in identifying and using aspects.

## Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.3 [Software Engineering]: Coding tools and techniques; D.2.13 [Software Engineering]: Reusable Software—*Domain engineering, Reusable libraries*

## Keywords

Aspect-oriented programming, AspectC++, Object-oriented frameworks

## 1. INTRODUCTION

Object-oriented frameworks are important for the development of large-scale industrial applications because they provide a common library of functionality (by defining an application programming interface or API) and an object-oriented model of the domain (as a set of class hierarchies that can be extended) [30]. Object-oriented frameworks typ-

ically provide classes that application developers can use directly, or reuse through object-oriented mechanisms such as composition, inheritance, and polymorphism [27]. Aspect-oriented programming can complement object-oriented programming by modularizing cross-cutting concerns that do not fit into inheritance hierarchies or procedural programming. As Kiczales et al. [17] state, “the central idea of AOP is that while the hierarchical modularly mechanisms of object-oriented languages are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems”.

We are investigating potential uses of aspect-oriented programming in the context of object-oriented C++ frameworks. The framework being studied is an object-oriented C++ framework used in the development of VLSI CAD applications at Hewlett-Packard. Existing framework-based C++ applications are of interest for two reasons. First, using existing applications can help understand the types of cross-cutting concerns that exist in legacy C++ software, which is often a mix of object-oriented and procedural styles. Second, maintenance of existing application software is a key part of the lifecycle of industrial applications, with practitioners reporting that it consumes more time and more resources than any other part of the software lifecycle [31]. We are studying multiple framework-based applications to identify common aspects used by many frameworks.

This paper reports on the results of identifying aspects in two framework-based applications. We have identified two types of aspects: framework-based aspects and application-specific aspects. Framework-based aspects modularize cross-cutting concerns that are based on not only the application structure, but also how an application uses the framework classes and API. Application-specific aspects represent cross-cutting concerns that exist in these C++ applications but do not use classes or methods of the object-oriented framework. For the aspects identified, we report on the benefits (code reduction, increased modularity, improved defect detection) as well as some potential drawbacks of refactoring applications to use aspects. Aspects that use framework calls or framework-based pointcuts can be grouped together as an aspect library that is associated with the framework.

The remainder of the paper is structured as follows. Section 2 describes the process used for identifying aspects for use in object-oriented frameworks. Section 3 describes the applications that were studied and the framework they are based upon. Section 4 details an aspect created for de-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD Industry Track 2006 Bonn, Germany  
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

bugging a framework-based application. In Section 5, we describe an extra-functional Timing aspect. Section 6 discusses refactoring a large application to use a policy enforcement aspect. A discussion of aspects and frameworks is provided in Section 7. Related work is reviewed in Section 8, followed by our conclusions in Section 9.

## 2. IDENTIFICATION OF ASPECTS

Aspects are typically associated with duplicated (or very similar) code that is scattered throughout an application and tangled with its local context [18]. These duplicates are often a good starting point for identifying framework-based aspects and application-specific aspects.

Framework-based aspects may also be based on framework extension points. These extension points include framework classes that applications commonly extend to form class hierarchies, callbacks that applications can use to register code to be called when certain events occur, and functors (function objects) that can be overridden to provide application-specific behavior in framework code. Framework documentation typically identifies expected extension points for application developers. Coady and Kiczales [6] reported improved modularity and maintainability when aspects were used to implement extension points in operating systems code. Since extension points are used in many applications and may be tangled with application-specific code, they may be good candidates for searching for framework-based aspects.

The design documents and source code of an application can be inspected for classes that are infrastructure-related (e.g., logging, memory management, performance monitoring), rather than directly related to the main functionality (VLSI CAD) of the application [14]. Since infrastructure-related classes typically affect many parts of an application, they are candidates for aspectualizing. Infrastructure-related aspects could be application-specific, or they could be framework-based if they modularize cross-cutting concerns related to how an application uses the framework.

Once we identify a concern that might be better modularized as an aspect, simple tools such as code browsers and Unix utilities like `grep` can be used to quickly find other occurrences of similar or related code to see if it is really cross-cutting. For example, if a logging class is used, we can `grep` all files for its type name (e.g. “`grep CadLogger *.cc`”). This has limitations, since `grep` considers one line at a time. For example, we cannot use `grep` to find occurrences of two class names within 3 lines of one another without using a complex script. In addition, `grep` is purely text based, while many code editors (e.g. Eclipse<sup>1</sup>, SlickEdit<sup>2</sup>) can display syntactic relationships for an entire project, such as class diagrams, function calls, and method calls. Visualizing more complex relationships can help identify methods that are called in many places or span hierarchies, and can also help determine if a concern is scattered enough to warrant creating an aspect.

Because of the large size and long weave-times associated with the framework-based applications we are studying, we have adopted a test-driven process that uses mock systems to prototype aspects before applying them to an application [25]. After prototyping an aspect with a small mock system,

we refactor the application itself and run existing application tests to ensure the change has the desired effect.

The refactoring process can be summarized with the following steps:

1. Review framework documentation for extension points.
2. Inspect application code for duplicate, tangled code.
3. Identify any infrastructure-related code that may cross-cut many applications.
4. For each aspect, use a small mock system to develop and test a prototype.
5. Refactor the application to use the aspect.
6. Run application regression tests.
7. To enable reuse, create abstract and concrete aspect.
8. Add framework-based aspects to aspect library.

Application-specific aspects also use the above steps, except for the first one. Since all of our applications are in the VLSI CAD domain, even application-specific aspects could be considered for library inclusion, since they may be usable by other applications in this domain.

Like traditional refactoring, the aspects are intended to preserve the behavior or intent of the program constructs [11]. The aspects will become part of a framework-based aspect library that we are developing. In order for aspects to be usable by other applications, an abstract aspect is created for the library. An abstract aspect has a virtual, abstract pointcut, and may have virtual methods that are called by the advice. The application uses a concrete extension of this aspect.

We have identified both production aspects and developmental aspects [19]. Developmental aspects are used for testing, profiling, tracing, or debugging an application, but are not part of the production system. Production aspects are required for the application to function properly and are delivered as part of the system.

## 3. APPLICATIONS AND FRAMEWORK

We have studied two applications: PowerAnalyzer and ErcChecker. We describe each briefly, since all aspects in this paper were created refactoring these applications. Both use an object-oriented VLSI CAD framework developed at Hewlett-Packard. The framework provides a class hierarchy for the domain of electric circuits, as well as parsers for common file formats, collection classes and iterators, and utility classes for performing common operations. The framework uses inheritance, including mixin classes, to provide many concrete classes and allow developers to create their own extension classes.

The PowerAnalyzer tool is used for estimating power dissipation of electrical circuits. It consists of about 12,000 lines of C++ code. Power dissipation results from switching power (signals changing from 0 to 1 or vice-versa) and from static leakage power (current that leaks away even when signals are not changing). The PowerAnalyzer tool is composed of 3 related executables: PowerSrc, PowerCap, and PowerEst. PowerSrc checks the structure of the circuit and ensures that all needed connectivity and electrical data is available. PowerCap is used to calculate capacitance since

<sup>1</sup>[www.eclipse.org](http://www.eclipse.org)

<sup>2</sup><http://www.slickedit.com/>

power switching power is proportional to capacitance. PowerEst reads data generated by the PowerSrc and PowerCap programs and generates a power estimate for each signal of a block. All three programs use an application-specific library of common functions and classes (`libPower`) in addition to using the HP VLSI framework. Because they are not variants of the same program, they are not a product line, but instead constitute a product family [4]. The use of several small, focused programs is a common design style for Unix programs [13].

Electrical Rules Checking (ERC) systems automate a number of electrical circuit checks. Example checks include checking for proper transistor ratios between the pull-up and pull-down transistors of an inverter, checking for fan-out limits, or checking for drive strength problems [26]. An ERC checking tool typically performs many types of checks on a circuit, reporting violations to the circuit designer. In order to understand (and ultimately correct) the violation, the user (a circuit designer) needs access to contextual circuit data, which the tool displays and writes to a log file. The `ErcChecker` tool consists of approximately 80,000 lines C++ code.

#### 4. A DEBUGGING ASPECT

The first aspect identified is a development aspect used in debugging the application's use of the framework. Validation and defect removal have been identified as challenges of using frameworks, since it can be difficult to determine if a fault is due to a defect in the application, unexpected interactions between framework code and application code, or a defect in the framework [27].

The `PowerAnalyzer` tool, like many VLSI CAD applications, invokes framework methods to create a large graph (based on framework classes) that represents circuit elements (transistors, capacitors), circuit connectivity between these elements (called nets or nodes), and other properties. It then populates the graph with additional data and manipulates it. Framework-provided iterators are used to traverse and explore the electrical circuit (also called a design).

During development of the `PowerAnalyzer`, the framework indicated there was an unrecoverable error due to incorrect internal framework state. The framework code printed the name of the framework method being called and exited. Calling `exit()` in a C++ library makes debugging difficult since program state information is not captured before exiting. Although the framework error message included the name of the method that exited, the application had many places where iterators were used. Further complicating this was the fact that the framework provides many types of iterators (e.g. iterate over all nets in a component, iterate over all instances in a component, iterate over all ports of a net) that share a common base class, and the error message was generated from the base class.

Initially, we used the brute force approach of adding print statements to record the last call before `exit()` was called. This required modifying 18 locations in 4 files, finding the defect and fixing it, and removing the 18 modifications from the 4 files. By contrast, the same effect is achieved by a single aspect in AspectC++. The `CadTrace` aspect, shown below, utilizes the fact that all the framework iterators provide a `Reset()` method that is called to start the iteration of elements when tracing all framework iterator uses by the application:

```
#include <iostream>
using namespace std;
aspect CadTrace {
    advice call("% %Iter::Reset(...)\")
    : before() {
        cerr << "about to call Iter::Reset for "
            << JoinPoint::signature() << endl;
    }
};
```

Because AspectC++ is a source-to-source translator, we created a separate `libPower` library that has the tracing enabled. Rather than having to unplug and plug in the aspect, we can link against the original code or the woven code to enable and disable tracing. Thus, if a similar bug is found later, we can simply switch to the traceable version of the library.

The `CadTrace` aspect depends only the framework: its pointcut and advice are not related to the application at all. Thus, it can be reused as is in any application that calls a framework `Iterator`. In addition, we can create an abstract tracing aspect that uses an abstract pointcut. Applications could then extend the abstract aspect and define the pointcut to match any framework methods.

#### 5. AN EXTRA-FUNCTIONAL TIMING ASPECT

Aspects have been proposed for modularizing extra-functional concerns: functionality that, while important, is not part of the central task of an application [24]. Because run-times for VLSI CAD software can be long (hours or even days), a common extra-functional concern is to write time stamps to a log file before certain steps or to write the elapsed time after certain steps. The `PowerAnalyzer` does this by implementing a class, `Timer`, that has a method to reset the elapsed time and another method to return the elapsed time as a string suitable for writing to a log file. Before and after various steps for which the designers want timestamp information, instances of `Timer` are created and used.

Even though similar code to instantiate and use `Timer` objects exists at many locations, there was no common structure or naming convention for use by pointcuts in selecting where an aspect should be woven. To refactor the `PowerAnalyzer`, functions that used the `Timer` class had the `Timer` instance and calls removed, and were renamed from `FunctionName` to `tmrFunctionName` for pointcut selectability. Another challenge was that some of the application code was written as large, procedural functions. One of the programs, `PowerEst`, contained a 500 line function with 15 separate uses of the `Timer` module. These separate uses were either loop statements or conceptually separate code blocks. For this function, the Extract Method refactoring was first used ("Turn the fragment into a method whose name explains the purpose of the method")[11]. Creating a function with name that begins with `tmr` allowed capturing the join point in AspectC++, and using a meaningful name allowed the join point signature to replace a manually generated text description of what was being timed. For consistency, the same aspect was applied across all three programs (`PowerSrc`, `PowerCap`, and `PowerEst`).

The `Timer` functionality is encapsulated in an aspect, `TimeEvent`, in which we use around advice to create a `Timer`

instance, call `Reset`, and then we proceed with the original call. After the original function invocation, we call `CheckTime` to get the current time as a string, and write this through a `PowerEstimator`-specific logging function (`PrintI`). We also call `PowerMessage::WriteBuffers()` to ensure that all files and logs are written out immediately and are not buffered by the operating system. The AspectC++ implementation is shown below:

```
aspect TimeEvent {

    pointcut outer_tmr() =
        call("% tmr%(...)") && !within("% tmr%(...)");

    advice outer_tmr() : around() {
        //set up the timer
        Timer timer;
        timer.Reset();
        //proceed with the original call
        tjp->proceed();
        //write out the time used
        PrintI(911, "Time around %s: (%s)\n",
            JoinPoint::signature(),
            timer.CheckTime());
        PowerMessage::WriteBuffers();
    }
};
```

This aspect represents a reduction in code of the following 4 lines of code that were normally duplicated:

```
Timer timer;
timer.Reset();
PrintI(911, "Time around %s: (%s)\n",
    JoinPoint::signature(),
    timer.CheckTime());
PowerMessage::WriteBuffers();
```

In the `PowerEst` program, long functions would reuse the same `Timer` object. The aspect-oriented solution used one `Timer` per `Reset` call. Since 15 occurrences were factored out, the reduction was  $15 \cdot 3 + 2 = 47$  lines of code. In place of manually using the `Timer` module, functions were created as pointcut targets. If creating a new function is thought of as adding 2 lines of code (one for the header or interface of the function, and one for the call), then 30 lines of code were added, for a total savings of 17 lines of code. In addition, one could argue that the resulting structure was also improved by using smaller, clearly named functions rather than a single monolithic function.

The `PowerCap` program, like `PowerEst`, contained a single monolithic function that used the `Timer` module seven times. It was also refactored into smaller functions. It had five locally defined `Timer` instances instead of only two, so the code savings when using the aspect was  $7 \cdot 3 + 5 = 26$ . Seven new functions were created, for an addition of 14 lines, resulting in an overall savings of 12 lines.

The `PowerSrc` program had only one function (`main`) that was timed, so it was refactored as:

```
int
tmrMain(int argc, char **argv, PowerMessage *pMsg);

int
main(int argc, char **argv)
```

```
{
    PowerMessage *pMsg = PowerMessage::GetMessage();

    return tmrMain(argc,argv, pMsg);
}
```

This change represents adding an extra layer of function call (3 lines) in `PowerSrc` (`main` calls `tmrMain`) and four lines of code are removed. Although the aspect only reduces one line of code in `PowerSrc`, using it ensures that all applications in the `PowerAnalyzer` product family use the `Timer` in a consistent manner.

For all three `PowerAnalyzer` programs the code savings was 30 lines. More importantly, all code for capturing and recording timer information is modularized into a single aspect. If the `Timer` were to be modified, or if a new timing module were substituted, the entire change could be made in the aspect and the underlying code would not be changed.

The style of name-based pointcuts results in tight coupling that could be inadvertently broken during maintenance due to name changes in functions [29]. Here, we have tight coupling between the `TimeEvent` aspect and the naming convention of methods. If a new function is added that should use the timer, it must begin with `tmr` or it will not have the `Timer` functionality woven in. In addition, if someone changes one of the names of the functions to no longer begin with `tmr`, time logging will no longer occur for that function. If a function that does not need timing information is created with a name that begins with `tmr`, that function will match the pointcut and have `TimeEvent` advice associated with it. Unfortunately, such problems would not be immediately detected since the regression tests for the `PowerEstimator` focus on functionality and not the non-functional, orthogonal concern of time logging [10].

One maintenance problem that we can avoid is a future direct use of the `Timer` module. Although AspectC++ does not provide a `declare error` construct like AspectJ's, C++ templates can be used to provide compile-time assertions by defining a template that has a boolean argument and only providing a definition for a true value. Code that instantiates the template with a false value will trigger a compile-time error [1].

Lohmann [21] suggested using a C++ template approach that can be combined with static join point information so that any calls to a function matching a pointcut will trigger an error. For the `TimeEvent` aspect, we can add the following advice so that any direct calls to the `Reset` method of the `Timer` class result in a compile-time error.

```
// create a template that has no definition
// when the second type value is 'false'
template <typename JP, bool>
    struct DirectCallOf__StartTimer;
template <typename JP>
    struct DirectCallOf__StartTimer<JP, true> {};

aspect TimeEvent {
    advice call("% Timer::Reset()") : before() {
        DirectCallOf__StartTimer< JoinPoint, false >();
    }
};
```

## 6. A POLICY ENFORCEMENT ASPECT

The `ErcChecker` implements 58 different electrical checks. Each one is a subclass of an abstract class `ErcQuery`, and must implement a set of common virtual methods, including `createQueries()` and `executeQuery()`. The `createQueries()` method is a static method similar to the Template Method design pattern [12], that is called on a particular circuit element, such as a transistor or electrical node. Since it is static, it is not associated with any single object; in this application, it is responsible for creating and evaluating objects of its class type. The `ErcChecker` iterates over various circuit elements, repeatedly calling `createQueries()`, which creates query instances and evaluates them by calling `executeQuery()`.

For each query, `createQuery()` performs the same sequence of conceptual steps.

1. Use framework iterators and framework traversal methods to identify circuit data needed by the query.
2. For each part of the circuit where the necessary circuit data is found, create an instance of the specific electrical query class associated with the check.
3. Call the `executeQuery()` method to on the query object from the step above. This method will indicate if the electrical check found an electrical failure or warning, or if the circuit element passed the electrical check.
4. Add queries that result in a failure or warning to a container class, the `LevelManager`, which generates electrical reports and can be used by a graphical user interface.
5. Write the results of `executeQuery()` to a log file.
6. Delete queries that did not result in a failure or warning.

Several of these steps can vary according to the particular query class. In step 2 for example, some queries create and evaluate multiple query objects for a single circuit element while others create only one query object. Step 4 can be modified with a command-line argument to the program so that all query results are reported instead of only warnings and failures.

Although conceptually similar to the Template method, no code is shared between classes, and there are significant variations between each class because of differences in the types of data being gathered. Thus, each of the 58 query classes has a single large method for `createQuery()`. In addition, for Query classes where many query objects are created, the six steps are repeated, one at a time, for each individual object, and some local context information is shared between the steps and accessed.

We created a `QueryPolicy` aspect to refactor steps 4 through 6. The aspect is implemented as after advice for the call to `executeQuery()`. Steps 1-3 are very query-specific, and are not refactored to an aspect. The AspectC++ implementation for the `QueryPolicy` aspect is shown below:

```
aspect QueryPolicy {
    pointcut exec_query(ErcQuery *query) =
        execution("% %::executeQuery(...)")
        && that(query);

    advice exec_query(query) : after(ErcQuery *query)
```

```
{
    if(gReportAll || query->errorGenerated()) {
        LevelManager::addQuery(query);
        gLog->log() << "Query error: "
            << " type: "
            << query->getName()
            << " start element: "
            << query->getStartName()
            << query->getSummary()
            << endmsg;
        query->logQueryDetails();
    }
    else {
        gLog->log() << "Query ok: "
            << query->getName()
            << endmsg;
        query->logQueryDetails();
        delete query;
    }
}
};
```

## 6.1 Specific Refactorings

While refactoring the `ErcChecker` to use the `QueryPolicy` aspect, several types of changes were made to adapt to the aspectual behavior. Many involved simple removal of code being handled by the aspect. We describe those that were not simple deletions of code below, followed by a summary of how many classes were associated with each type of change.

### 6.1.1 Error-only queries

Before refactoring scattered code to an aspect, some classes had a much simpler `createQueries()` structure than others. For example, most queries could pass or fail, so the query had to call the `errorGenerated()` method after calling `executeQuery()`.

```
query->executeQuery();
if(gReportAll || query->errorGenerated()) {
    LevelManager::addQuery(query);
    gLog->log() << "Query error " ...
}
else {
    gLog->log() << "Query ok: " ...
    delete query;
}
```

For some query classes, if a particular circuit structure was found at all, it always indicated the presence of an electrical error. The query code was simpler, since it did need to check the result of `query->errorGenerated()` and can immediately add the query object to the `LevelManager` class, as shown by these 3 lines:

```
query->executeQuery();
LevelManager::addQuery(query);
gLog->log() << "Query error " ...
```

The simpler structure has the potential for a defect in the refactored system. The `query->errorGenerated()` method uses an object attribute that should be set by `executeQuery()`. If the `executeQuery()` method fails to set that attribute, the original code (which did not check it) would function properly, but the new, aspect-based code, which uses advice that always checks the attribute value would

not function properly. These classes were inspected to ensure that their respective `executeQuery()` methods set the required attribute. Thus, although refactoring to an aspect did not introduce a defect, it does illustrate a case where using an aspect's advice with many core concern classes could introduce a defect if methods in a class heirarchy do not consistently use state variables.

### 6.1.2 Multiple queries in `createQueries()`

For most queries, `createQueries()` creates and evaluates a single query object. For others, exactly two objects are created (one for the high voltage case, and one for low voltage). In addition, some query classes have the potential to find many possible violations from a single starting point and iterate over a variable number of cases. For example, a wire in a circuit may have many neighbor nets with which it shares an unacceptable amount of coupling capacitance. When refactoring a class to use the `QueryPolicy` aspect, we must ensure that all of the calls to `executeQuery()` are removed and handled instead by the aspect.

### 6.1.3 Query-specific logging

In addition to the standard query logging, some queries contain additional class-specific code that calls methods not inherited from the base class to record query-specific details used for validating and debugging the electrical checks being performed. Because this is class-specific code, it cannot be called from an aspect advice that uses only the base class interface.

This code was refactored into a new method, `logQueryDetails()`, and directly called by the policy aspect. The base class for the queries was modified to provide an empty default implementation for queries that do not need this functionality. For classes that did need the functionality, the class had to define the `logQueryDetails()` method.

In the original code for classes that had query-specific logging, the code was structured like this:

```
query->executeQuery();
    if(gReportAll || query->errorGenerated()) {
        LevelManager::addQuery(query);
        gLog->log() << "Query error " ...
    }
    else {
        gLog->log() << "Query ok: " ...
        delete query;
    }
    /* QUERY-SPECIFIC CODE HERE,
       USING 'query' VARIABLE */
    if(!(gReportAll || query->errorGenerated()))
        delete query;
```

The call to `logQueryDetails()` must be performed in the aspect, because if we leave the query-specific code in (represented by the comments in all caps), and then use the aspect, we end up with this structure:

```
query->executeQuery();
/* Aspect will be called here */
/* QUERY-SPECIFIC CODE HERE,
   USING 'query' VARIABLE */
```

At first glance, we see the reduction in code that the aspect provides, but we may not realize that we could have a problem. If the query does not indicate an error, then the

aspect's advice will delete the query object. In that case, when control returns to the method, the query-specific logging code will make method calls with the `query` variable. Invoking a method call on a deleted object in C++ will result in erroneous program termination. By having the advice call `logQueryDetails()` before the object may be deleted, we will not have method calls on deleted objects.

Since adding an empty method to the base class is a one line change, we added the method directly to the C++ header file rather than using an AspectC++ introduction. Creating `logQueryDetails()` required copying and pasting the query-specific code into the method body and changing method invocations to intra-class calls. Thus, method calls on an object like this:

```
gLog->log() << " Coupling net: "
           << query->getCouplingNet()
           << " capacitance: "
           << query->getCouplingCap() << endmsg;
```

are now part of the `logQueryDetails()` method that has an implicit query object:

```
void HighCouplingCap::logQueryDetails()
{
    gLog->log() << " Coupling net: "
              << getCouplingNet()
              << " capacitance: "
              << getCouplingCap() << endmsg;
}
```

### 6.1.4 Missing log information

The `QueryPolicy` aspect is woven into all calls to `executeQuery()`, consistently applying the policy through advice. During refactoring, we found that one query failed to provide any information to the log file, but was otherwise correct. Changing to the aspect fixed this oversight, eliminating an existing subtle defect.

### 6.1.5 Unchanged queries

There were a number of queries that were not electrical rules queries, but instead were used to flag non-electrical failures, such as problems in the system itself (e.g., failure to open a log file and unexpected data structure value in the framework). Warning a circuit designer of these problems was accomplished by implementing them as sub-classes of the `ErcQuery` base class. Doing this allowed the system error queries to be written to error files and displayed in the graphical interface consistently with the electrical checks, leveraging existing functionality. The non-electrical error classes did not actually call the `executeQuery` method, but directly added objects to the system when system-related issues were found during a tool run.

Because these system error classes differed from the electrical query classes, the non-electrical classes were not modified to use the `QueryPolicy` aspect. Since the aspect uses the `executeQuery` method, which is never called for these, the aspect did not affect them at all. The presence of these classes reflects a sub-optimal design decision that fortunately was easy to work around. A better long-term change might be to extract a subclass that distinguishes between family of electrical queries and the various non-electrical system queries [11]. Legacy systems often have methods or classes that could be refactored for a cleaner design, as

demonstrated by the large number of documented refactorings. In general, sub-optimal design decisions could affect aspect design and pointcut selection.

## 6.2 Summary of changes

The types of changes made (as well as the simple case of only deleting code) are shown in Table 1. Because some of the 58 queries involved more than one change type, the numbers in this table add up to more than the number of query classes when the number of unchanged queries (13) from section 6.1.5 are taken into account.

Type of change	# Queries
Simple deletion of code	15
6.1.1 Error-only queries	8
6.1.2 Multiple queries in createQueries()	6
6.1.3 Query-specific logging	18
6.1.4 Missing log information	1

Table 1: Changes made for QueryPolicy aspect

## 6.3 System-wide challenges

In addition to specific query-related challenges encountered during refactoring, we also found three system-wide issues that affected correctness, testability, and the change process.

### 6.3.1 Accidental code duplication between aspects and the core concern

The QueryPolicy aspect deletes query objects that do not find electrical errors. In refactoring the code, the original calls to `delete` must be factored out, or both the advice code and query code will try to delete the same object. This would introduce a defect that results in program termination. The underlying issue here is that aspect-oriented refactoring is asymmetric, since the duplicate scattered (such as `delete`) must be manually removed, but the corresponding call to `delete` in the advice will be automatically woven in.

Debugging the root cause for multiple deletions of the same object requires understanding the interaction between the aspect and core concern. If traditional tools such as debuggers are used, code comprehension may be more difficult, since the developer will be examining the woven code, which contains core concern code, aspect code, and low-level constructs used by the AspectC++ implementation of an aspect.

### 6.3.2 Standardizing output and regression testing

The QueryPolicy aspect standardizes the log file messages that indicate pass or fail for each query. We consider this a benefit for long-term maintenance, but it does cause any regression tests that look at output logs to fail due to the (now standardized) output. For this application, only a few more than 100 regression tests look at the log files; most look at report files that were not changed. In general, this kind of one-time maintenance change can present a significant adoption cost for AOP on large projects with extensive test suites.

### 6.3.3 Compile times

For the ErcChecker, weaving and compiling the code takes 25 minutes. This makes developing aspects, checking for

proper pointcut matches, and testing advice code a tedious process. In fact, the desire for a faster, iterative prototyping approach to be able to experiment with and validate aspects was part of the motivation for our use of mock systems and test-driven development [25]. By creating a small mock system that had the same basic query structure and naming conventions, we could experiment with different pointcut statements and advice.

## 6.4 Benefits

Using the QueryPolicy aspect represents a line reduction of about 8 lines per class. However, as discussed in Section 6.1, not all classes had the same number of lines removed. In addition, factoring out `logQueryDetails()` added one line to the base class. Each class that used it needed to define the method in its C++ header file, and then the method name, opening curly brace (`{`) and closing curly brace (`}`) added 3 more lines. Thus, this change added  $1 + 4 * 18 = 73$  lines of simple boilerplate code. The total difference in lines for the changes was a reduction of 262 lines. Taking into account the additional lines added for `logQueryDetails()`, 335 lines were removed from 45 classes. The aspect code that replaces this code is only 21 lines long.

More important than the lines of code reduced, the aspect enforces a policy of what must *always* occur after `executeQuery()` is called. During development, there were cases where the policy was accidentally missed for a class, since it cannot be automatically statically enforced when scattered through the many `createQuery()` methods for the subclasses.

## 7. DISCUSSION

Using the two framework-based applications, three aspects were identified. The first, a developmental aspect, was used to trace calls into the underlying framework. We are also investigating the identification of production framework-based aspects.

The other two aspects are production aspects, but there were still some important differences. The Timer module in the PowerAnalyzer can malfunction (or fail to run) without invalidating the results of the tool. By contrast, the policy enforcement aspect for the ErcChecker is required for proper functioning of the tool.

Other developers who use the same CAD framework have been enthusiastic about using development aspects, but have expressed concern about migrating to a new paradigm and a dependency on an aspect weaver for production aspects. Evaluating the use of aspects on these large C++ applications is beneficial for understanding the possible costs, benefits, and risks of adopting a new technology. Migrating to new technologies is not without risk and needs to be done carefully. In fact, VLSI CAD software saw an early unsuccessful attempt at large-scale object-oriented programming in the mid-1980s which adversely impacted the schedule and performance of a market leader's products [20]. In addition to demonstrating benefits and costs of using aspects, studying AOP in the context of large-scale applications can help provide a smooth transition in industrial use.

## 8. RELATED WORK

### 8.1 AOP-based Refactoring

Advocates of aspect-oriented programming have begun enumerating AOP-based refactorings and evaluating the associated benefits and costs [18]. Coady and Kiczales [6] demonstrated the benefits of using aspects in operating system code by implementing four modules as aspects in an early version of FreeBSD and then observing the changes to those modules as they introduced the changes from two subsequent versions. Our work focuses on framework-based applications, and has the long-term goal of studying a set of applications for common aspects among them.

## 8.2 Product Lines

Batory et al. [2] propose replacing large framework hierarchies with a set of components that can be combined in a layered approach to build “product lines” – families of related applications. Mixin classes (multiple inheritance) and templates based on custom-designed flexible component classes are combined using a grammar-based approach that specifies compositions [3]. Their product line approach relies on completely replacing (or reimplementing) the framework with small components that can be composed in pre-defined, grammar-based ways across many layers. Our approach uses an existing object-oriented framework without modifying it. Instead, the framework-based applications are modified to use aspects.

Lohmann et al. [24] propose the use of domain analysis, which produces feature diagrams, which are then used to guide in the designing an ‘architecture-neutral’ system that will allow aspects to be woven in across multiple modules so that the non-functional properties can be configured across a set of product lines. Although we may use domain analysis to identify non-functional aspects, our approach deals with a set of applications that share a common framework. Some of the applications we are refactoring are implemented as product families; however, the overall set of applications are not a product line since they are not variations of the same functionality.

## 8.3 Long-term studies of aspect-oriented refactoring

Coady [6, 7, 8, 9] has studied how aspect-oriented programming can be used to refactor complex code for operating systems. Included in her work was a retroactive study where aspects were added to an early version of the operating system code, and then the changes were applied to the refactored version to see how well-suited aspects were for system evolution. We plan on carrying out a similar longitudinal study, but will do so for a set of framework-based applications that use an aspect library.

## 8.4 C++ Templates and Obliviousness

C++ provides a powerful template mechanism that can be used to generate classes for types at compile time. Alexandrescu [1] shows how policy classes in C++ can use templates to provide structures that, like aspects, have an interface but still allow users a means to extend the internal code structure. Lohmann, Gal, and Spinczyk [23] demonstrate that these mechanisms can be used to develop code with an aspect-oriented style, but without the obliviousness of aspects: everything must be explicitly instantiated through templates, which have to have the extension points designed in.

The main limitation of implementing aspects as templates

and of policy classes is that the extension points of an aspect or the actual policy location of a policy class must be designed into the template [23]. By contrast, obliviousness in aspect-oriented programming can be used to extend and customize classes and frameworks in ways they were not explicitly designed for. The use of AspectC++ with templates has been explored by Lohmann, Blaschke, and Spinczyk [22].

Burke [5] points out that obliviousness allows needed orthogonal behavior to be layered in above or below the functionality that the behavior crosscuts so that the layered functionality can easily be enabled or disabled as needed. In our approach, we only weave aspects into the applications and not the framework code. The layers used by our aspect library would be around the framework and inside the applications but not below the framework interface.

Recently, obliviousness has been criticized as inadequate when designing and implementing new systems, since it requires sequentialization of the process; that is, first the base code is developed, and then the aspects are written based on syntactical properties and structure of the base code. Sullivan et al. [28] propose instead to define an interface between the base code and aspects, so that both can be evolved without accidental changes in dependencies. When using aspects with existing frameworks, however, we believe that sequentialization is fine: the framework is already implemented and available, and applications are developed to utilize the existing framework.

## 9. CONCLUSIONS AND FUTURE WORK

This paper has presented initial work at identifying and using aspects in framework-based VLSI CAD software written in C++. Both development and production aspects have been identified and used. The initial results in terms of reduction of code and modularity (grouping together related code) show improvement with an aspect-oriented approach.

Future work will continue investigating identifying aspects and performing refactoring of framework-based applications. We have begun investigating clone detection tools such as CCFinder<sup>3</sup> [16] for finding concerns, and are also interested in aspect-mining tools. Unfortunately, many current aspect mining tools are based on Java rather than C++ [15].

In addition, we are interested in identifying development and production aspects that are based on the framework so that a library of framework-based aspects can be created and used with many framework-based applications. We believe this approach, which we refer to as aspectualizing a framework, allows aspects to enhance framework functionality. In addition, it will ease integration with and use of frameworks without modifying the framework itself.

## 10. ACKNOWLEDGMENTS

The authors wish to acknowledge the use of the open source tool AspectC++ (www.aspectc.org). They also acknowledge helpful feedback on the AspectC++ user group from Olaf Spinczyk and Daniel Lohmann.

## 11. REFERENCES

- [1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. AW C++ in Depth Series. Addison Wesley, January 2001.

<sup>3</sup>CCFinder v10.1.2

- [2] D. Batory, R. Cardone, and Y. Smaragdakis. Object-oriented frameworks and product lines. In *1st Software Product-Lines Conference (SPLC1)*, 2000.
- [3] D. Batory and B. J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, Feb. 1997.
- [4] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *ASE '02: Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, page 81, Washington, DC, USA, 2002. IEEE Computer Society.
- [5] B. Burke. It's the aspects. *Java Developer's Journal*, 2003.
- [6] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 50–59. ACM Press, Mar. 2003.
- [7] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring operating system aspects: using AOP to improve OS structure modularity. *Commun. ACM*, 44(10):79–82, Oct. 2001.
- [8] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *ACM SIGSOFT Software Engineering Notes*, 26(5):88–98, Sept. 2001. Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT symposium on Foundations of software engineering, Vienna, Austria.
- [9] Y. Coady, G. Kiczales, J. S. Ong, A. Warfield, and M. Feeley. Brittle Systems will Break - Not Bend: Can AOP Help? . In *Proceedings of the 10th ACM SIGOPS European Workshop on Operating Systems*. ACM Press, September 2002.
- [10] A. Colyer, A. Rashid, and G. Blair. On the separation of concerns in program families. *Technical report, Lancaster, COMP-001-2004*, 2004.
- [11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.
- [13] M. Gancarz. *The UNIX Philosophy*. Digital Press, December 1994.
- [14] S. Ghosh, R. B. France, D. M. Simmonds, A. Bare, B. Kamalakar, R. P. Shankar, G. Tandon, P. Vile, and S. Yin. A middleware transparent approach to developing distributed applications. *Software Practice and Experience*, 35(12):1131–1154, October 2005.
- [15] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition in legacy code. In P. Tarr and H. Ossher, editors, *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*, May 2001.
- [16] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.
- [17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.
- [18] R. Laddad. Aspect-oriented refactoring part 1: Overview and process. Technical report, TheServerSide.com, 2003.
- [19] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.
- [20] J. Lakos. *Large-Scale C++ Software Design*, chapter 0, pages 1–18. Addison Wesley, July 1996.
- [21] D. Lohmann. 2006. AspectC++ user's mail list: <http://www.aspectc.org/pipermail/aspectc-user/2006-January/000872.html>.
- [22] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*. ACM, 2004.
- [23] D. Lohmann, A. Gal, and O. Spinczyk. Aspect-Oriented Programming with C++ and AspectC++ (Tutorial). In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*. ACM Press, Mar. 2004.
- [24] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In D. H. Lorenz and Y. Coady, editors, *ACP4IS: Aspects, Components, and Patterns for Infrastructure Software*, Mar. 2005.
- [25] M. Mortensen, S. Ghosh, and J. Bieman. A test driven approach for aspectualizing legacy systems. 2006. Submitted to Aspect-Oriented Software Development (edited by Sami Beydeda and Volker Gruhn).
- [26] S. M. Rubin. *Computer Aids for VLSI Design*. Addison-Wesley VLSI Systems Series. Addison-Wesley, 1987.
- [27] D. Schmidt and M. Fayad. Object-oriented application frameworks. *Communications of the ACM*, 10(40):32–38, 1997.
- [28] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewan, and H. Rajan. On the criteria to be used in decomposing systems into aspects. In *In European Software Engineering Conference and International Symposium on the Foundations of Software Engineering*, 2005.
- [29] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.
- [30] J. van Gurp and J. Bosch. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Software Practice & Experience*, 10(31):277–300, October 2001.
- [31] J. P. Zagal, R. S. Ahus, and M. N. Voehl. Maintenance-oriented design and development: A case study. *IEEE Software*, 19(4):100–106, 2002.