Detecting Design Defects Caused by Design Rule Violations

Sunny Wong, Yuanfang Cai, and Michael Dalton
Drexel University
Philadelphia, PA, USA
{sunny, yfcai, mcd45}@cs.drexel.edu

ABSTRACT

Research has shown that unintended violations of dependencies can incur modularity decay, higher maintenance cost, and software defects. However, dependency violations may be undetected by traditional software verification and validation techniques. In this paper, we present an approach to detect potential design defects caused by the violation of *design rules*. We identify the patterns of how different components change together to implement modification requests, and use a logical model to predict change scope according to the design rule theory. We then calculate the differences between the predicted and actual co-change patterns to reveal potential defects. To evaluate our approach, we investigated the version history of 15 releases of Hadoop and 9 releases of Derby. Our approach revealed such issues as poorly designed inheritance hierarchies, tightly coupled cyclical dependencies, and code clones. Some problems we identified were indeed refactored in later versions, validating the effectiveness of our approach.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*design defect*

General Terms

design defect, design rule theory

Keywords

design defect detection, design rule theory

1. INTRODUCTION

Empirical studies have revealed a strong correlation between software defects and unintended violations of dependencies [10, 12, 19, 23]. Numerous approaches (e.g. Fenton and Neil [15]) have been proposed to predict the number or the density of software defects related to dependency structures [19, 23]. The problem is that these approaches do not allow software designers to detect and reveal concrete defects, early at the development stages, or as soon as the defects occur.

For example, our recent work showed that inexperienced developers may forget to remove experimental scaffolding code that should not be kept in the final product, and an application programming interface (API) may be accidently defined using non-API classes [21]. These dependency violations are not easily detectible by traditional software verification and validation techniques because they may not influence the functionality of the system but will cause modularity decay and lead to expensive system-wise refactoring.

In this paper, we present an approach to detect such defects between software releases. First, we find co-change patterns that reveal how components change together to fulfill modification requests. Second, we model the original design, that is, the version prior to these changes, using a logical model called the *augmented constraint network* (ACN) [7,9]. From this ACN model, we predict how components *should* change together based on Baldwin and Clark's design rule theory [3]. Given the *predicted* change scopes and the *actual* co-change patterns, our approach then computes the differences and thus reveals potential design defects.

Our rationale of using design rule theory [3] as the basis of design defect detection is as follows. Design rules, defined as stable design decisions that decouple a system into independent modules, determine the modular structure of a software system. For example, APIs, naming conventions, or design patterns can be seen as design rules that, once stabilized, will enable subordinate modules to be changed freely without affecting each other. When making high-level design-rule decisions, such as choosing an architectural style or designing an inheritance hierarchy, designers make implicit assumptions regarding which parts of the system may change and which parts will remain stable.

When software evolves, however, it may or may not change as the designers originally assumed. If such discrepancies exist, the software may be changed in unpredicted ways, deviating from the original modular structure determined by design rules: the modules that were designed to be independent may have to always change together, and design rules that are supposed to be stable may have to be touched frequently. For example, if changes to class `B` often requires changes to its parent class `A` then the design may be problematically unstable, because the impact on `A` may cause a ripple-effect to its other children. Class `A` is a *design rule* by Baldwin and Clark's definition; it should not be affected by its subordinate design decisions (its sub-classes).

We observe that, by investigating how components *should* change together according to the design rule theory versus how components *actually* change together in reality, there is a potential to reveal these design defects. Considering the above example, if we can predict that `A` should not change because of changes to `B`, but the change history shows the opposite, we know there is a problem: either the reality deviates from the designer's assumptions, or suspicious dependencies were introduced, e.g. through quick and dirty maintenance activities. In these cases, the modular structure framed by the original design rules is deviated. We call such a deviation as a *design rule violation*.

In Section 3, we present our algorithm for predicting how components *should* change together using ACN modeling, which formalizes the key concepts of the design rule theory. Different from our original ACN-based design change impact analysis technique [7, 8] that relies upon constraint-solving and lists *all* possibly impacted decisions, this algorithm does not require constraint-solving and generates only a few predictions based on the level of importance and involvement of design rules. Different from our recent short paper [29] that reports the results of a change impact prediction method that aims to improve the accuracy of prediction, the algorithm presented in this paper aims to reveal the discrepancy between prediction and reality, which may generate highly inaccurate predictions.

To identify co-change patterns from a version history, we leverage some concepts and techniques used in state-of-the-art change impact analysis techniques, which mine revision history or modification requests to figure out how different parts change together in the past [31, 32]. Similar to our recent work [29], their purpose is to increase the accuracy of impact scope prediction [4, 5]. However, different parts of a system can change together because of design defects or erroneous dependencies. For example, cloned code may have to be changde together, or, two sub-classes of a parent class may always unexpectedly change together if their common parent class is not defined correctly.

Without the luxury of observing the effectiveness of this approach in ongoing projects, we leverage the version histories of open source software systems to evaluate our approach in a retrospective way: from the revision history records, we identify co-change patterns; for each release, we generate its ACN model automatically and predict how components should change together. After that, we identify *potential defects* by calculating the differences. We evaluate our approach (Section 4) using 15 versions of Hadoop Common[1] and 9 versions of Derby[2]. Since we are not the designers of these systems, given any potential defect calculated by our approach, we have to manually examine the system to determine if it is really a defect or a false positive.

We chose these systems first because they employ effective bug tracking systems that allow us to collect detailed information about how modification requests are fulfilled. These systems also have relatively short version histories and are shown to be well-modularized measured using existing modularity metrics. This means that we will not be overwhelmed by a huge number of potential defects. Moreover, still being able to identify important design defects in software systems shown to be already well-modularized by existing metrics proves the potential utility of our approach from another perspective.

In the 15 versions of Hadoop and 9 versions of Derby, our approach revealed 4 potentially important design defects for each system, at various points of their version histories. Many of those defects were repeatedly revealed in multiple versions. Of the total 8 potential defects, two of them that were first identified in early versions were indeed refactored in later versions, strongly verified our results. Four other potential defects revealed tightly coupled cyclical dependencies that could and should be replaced with better designs that are obviously available. We have not seen these parts being refactored in their history, but two of them were identified in the latest versions. For the rest of the potential defects, since we lack the domain knowledge to determine for sure whether they are really defects or not, we classify them as false positive results.

In the next section (Section 2), we introduce the following concepts needed to understand our approach: the design rule theory, the ACN modeling and its decomposition, the transformation from UML class diagram to ACN used in our evaluation, and the data mining concepts used in co-change pattern identification. Section 5 discusses our results. Section 6 presents related work and Section 7 concludes.

2. BACKGROUND

In this section, we present background information on Baldwin and Clark's design rule (DR) theory [3], augmented constraint network (ACN) modeling, and co-change pattern mining concepts. These theories and concepts provide the foundation for understanding our approach and evaluation procedure.

2.1 DR Theory and ACN Modeling

Figure 1 shows a UML class diagram of a (partial) class hierarchy from the Hadoop system that we will discuss in Section 4. This diagram shows a generic `FileSystem` class and two specific types of file systems, the `LocalFileSystem` and `DistributedFileSystem`. Diagrams like this can be easily reverse engineered from source code with existing tools. The UML diagram implies a number of assumptions and constraints among these classes. For example, the `FileSystem` class is assumed to be stable and unlikely to change because changing it may impact its multiple sub-classes. In Baldwin and Clark's modularity theory [3], this class is a *deign rule*, defined as a stable design decision that decouples otherwise coupled modules and creates independent modules. Ideally, design rules should be not be impacted by changes to decisions that are subordinate to them. For example, changes to the class `DistributedFileSystem` should not force the change to `FileSystem`, otherwise other sub-classes may be affected. These are the types of design rule violations that we target to detect with our approach.

We developed a logical framework called the *augmented constraint network* (ACN) that formalizes the concept of design rules. Figure 2 shows part of the ACN derived from the UML diagram shown in Figure 1 (for brevity, class names have been abbreviated in this ACN). An ACN consists of a *constraint network* that models design decisions and their relations, a *dominance relation* that formalizes design rules, and a *cluster set* in which each cluster represents a different way to partition a design (not used in this paper).

[1]http://hadoop.apache.org/common/
[2]http://db.apache.org/derby/

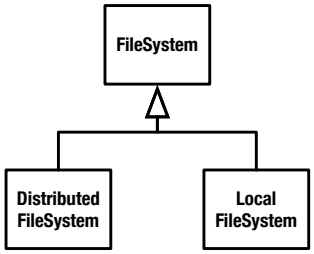

Figure 1: FileSystem Inheritance Hierarchy

```
1.  FS_interface : {orig, other};
2.  FS_impl : {orig, other};
3.  LFS_interface : {orig, other};
4.  LFS_impl : {orig, other};
5.  DFS_interface : {orig, other};
6.  DFS_impl : {orig, other};
7.  FS_impl = orig => FS_interface = orig;
8.  LFS_impl = orig => LFS_interface = orig;
9.  DFS_impl = orig => DFS_interface = orig;
10. LFS_interface = orig => FS_interface = orig;
11. LFS_impl = orig => FS_interface = orig;
12. DFS_interface = orig => FS_interface = orig;
13. DFS_impl = orig => FS_interface = orig;
14. (FS_impl, FS_interface);
15. (LFS_impl, LFS_interface);
16. (DFS_impl, DFS_interface);
17. (LFS_interface, FS_interface);
18. (LFS_impl, FS_interface);
19. (DFS_interface, FS_interface);
20. (DFS_impl, FS_interface);
```

Figure 2: Augmented Constraint Network

A constraint network consists of a set of design *variables*, which model design dimensions (or relevant environment conditions) and their domains; and a set of logical *constraints*, which model the relations among variables. The main relations we model using an ACN is the *assumption* relation among design variables. In Figure 2, lines 1–6 are the variables modeling the class hierarchy, and lines 7–13 are the constraints. For example, line 11 models that the implementation of the `LocalFileSystem` class assumes that the interface of the `FileSystem` is as originally agreed.

We augment the constraint network with a binary *dominance relation* to model asymmetric dependence relations among decisions, the essence of design rules, as shown in lines 14–20. For example, line 14 indicates that the decision for how to implement the `FileSystem` class cannot influence the design of its interface; in other words, we cannot arbitrarily change the `FileSystem` class's public interface to simplify the class's implementation because other components (including its sub-classes) rely on that interface.

Cai and Sullivan [9] showed that an ACN could be decomposed into sub-ACNs such that each sub-ACN represents all the decisions needed to realize a feature in the system. Their algorithm to accomplish this starts by building a condensation graph of the ACN. To build the condensation graph, we first create a directed graph with each vertex modeling a variable in the ACN, and edges (in both directions) between the variables in the same constraint. Then edges are removed based on the dominance relation and vertices are collapsed into strongly-connected components. They observed that each minimal element of the condensation graph represents a feature, and all the chains ending with a minimal element contain all the decisions needed to realize the feature. Our design-rule based prediction algorithm, which we describe in Section 3.2, leverages the condensation graph and the ACN decomposition algorithm.

Due to the steep learning curve of constructing ACNs in practice, we recently developed techniques to automatically convert UML class diagrams [29] and component diagrams [24] into ACN models. The basic idea is to model each class with two ACN variables, representing its public interface (with a name ending with `_interface`) and private implementation details (with a name ending with `_impl`). Constraints and dominance relation pairs are automatically defined based on the UML relations. The ACN shown in Figure 2 was automatically derived using our conversion technique. For the evaluation of our defect detection technique, we reverse engineered UML class diagrams from Java code bases and used this technique to obtain ACN models as input to our defect detection algorithm.

2.2 Co-Change Pattern Mining

Since part of our approach involves identifying co-change patterns, similar to the recent work of predicting change impact scope from version history, we define some terms and briefly describe concepts from the topic. We use the following terms throughout this paper, consistent with the definitions of Ying et al. [31]. A *modification task* can be either a bug fix or enhancement task. A *solution* is the set of of files that contribute to an implementation of a modification task. The cardinality of a solution is the number of files it contains. A *change source* is a class that developer will initially change, for which the impact scope is to be identified. Arnold and Bohner [2] refer to the change source as the starting impact set. The terms *transaction* and *change set* are used interchangeably to mean a set of files that are committed together into revision control.

To identify co-change patterns for impact scope prediction, the revision history is mined and files that are changed within the same transaction are recorded. The more often two files are changed within the same transaction, the more coupled they are, in terms of *logical dependencies*. Pattern-based prediction techniques, like the work of Ying et al. [31], set a minimum number of times two files must be changed together before one is predicted in the impact scope of changing the other (minimum support). Other approaches, like the work of Zimmermann et al. [32], require not only a minimum absolute number of transactions but also a minimum percentage of transactions (minimum confidence). In order words, if file A appears in 5 transactions with file B but file B appears in 45 transactions without A, then the confidence that A should be recommended for the impact scope of B is $5/50 = 0.1$. For our evaluation, we only use support for identifying co-change patterns. Note that the definitions of support and confidence used in co-change pattern mining is very similar to the definition of support and confidence that we define later, as heuristics for identifying potential defects. We purposely chose to use the same terms due to their similarity.

3. APPROACH

The approach proposed in this paper relies upon the existence of detailed and abundant information about modification tasks and their solution. As a result, our approach should be applied at a point where this information is available. Figure 3 shows one possible point where our approach can fit in traditional software life cycle. This figure roughly shows three commonly used phases of a release cycle: *Develop*—designing and developing the code, *Test*—testing functional and non-functional requirements, and *Release*—releasing the result to customers. Our approach, which we call a *Refactor* phase, can be added prior to development of the next release when all the modification tasks are completed and tested for the just released version.

Our approach is independent from traditional testing phase, and the purpose is not to identify functional defects. By introducing this phase prior to the next development phase, we have the complete set of modification tasks and tested solutions for the most recent release. Applying our approach at this point will only detect structural defects. Fixing these defects will ensure that the next release is developed upon a better structured software basis. The *Refactor* phase can also be applied immediately before the *Release* phase as long as all the modification tasks are completed and tested. In the next subsections, we give an overview of the *Refactor* phase and detail the design-rule-based change scope estimation algorithm.

3.1 Overview

We use modification task #51 from Hadoop, based on the version 0.1 code base towards release 0.2, entitled "per-file replication counts," as an example to explain the defect detection approach in the *Refactor* phase, as shown on the right side of Figure 3. The transaction commit message states "Support per-file replication counts in DFS." From this message, we identified the change source to be the `DistributedFileSystem` (DFS) class. The solution resolving this modification task consists of twelve source code files and two other files.

From the source code of version 0.1, we first derive an ACN using the approach briefly described in Section 2 and detailed in our previous work [20], and predict the impact scope triggered by changes to DFS using the algorithm to be introduced in the next subsection, as shown in the first row of the Figure. In the meanwhile, we identify what other classes generally change together with DFS, by finding the co-change pattern with DFS as the change source and looking at how frequently classes change together, denoted in Figure 3 as *Co-Change Pattern*.

Given a modification task and its solution, we first compare the co-change pattern with the actual solution, and take the intersection of these two sets, which reflects the occurrence of the pattern in the particular solution. After that, we compare the resulting set with the ACN-predicted impact scope. The differences between the sets reflect the discrepancy between the predicted impact scope and actual co-change patterns, hence the location of potential defects. We name these set operations in Figure 3 as *Clio* (pronounced as klee-oh). We call the resulting set and the change source together as a *candidate defect*. For our example DFS modification request, the candidate defect consists of `DistributedFileSystem`, `FileSystem`, `LocalFileSystem`, and `FSNamesystem`. Two versions later, in modification request #318, we observed that the exact same candidate defect, with `DistributedFileSystem` as the change source, occurred twice. The repeated appearance of this candidate defect increases the probability that it is a real defect.

We conjecture that the more frequent a candidate defect is detected, the more likely that it actually is a design defect. The frequency also depends on such factors as the length of version history. If a candidate defect only appears once in a long version history, it may not be a real defect. We use two heuristics, *support* and *confidence*, to assess the probability that a candidate defect is a true defect, similar to Zimmermann et al's [32] definition for identifying *rules* to predict impact scope from version history. Given a change source x and component y, we define the *support* that there is a defect involving x and y to be the number of modification requests where x is a change source and y is involved in its candidate defect. Considering the DFS example, by version 0.3, our approach identified 3 occurrences of the candidate defect, so its support level is 3. The *confidence* determines the *strength* of the conclusion. Using the same example, since DFS is involved in 3 modification requests and `FileSystem` (FS) is involved in all these candidate defects, the confidence that DFS and FS are involved in a design defect is 1.

The minimum support and confidence values, based on which an alarm should be raised and investigations should be made to see if there is a real design defect, can only be empirically based on the age and length of existing version history. We call a *candidate defect* with sufficient *support* and *confidence* as a *potential defect*. Given a potential defect, the designer can further investigate these components to see if there exists a real design defect. Our approach only calculates and reveals potential defects; determining whether they are true defects may depend on domain knowledge.

3.2 DR-based Impact Scope Prediction

In this subsection, we describe our approach of predicting how components *should* change together based on ACN modeling and the design rule theory. Our algorithm starts from an ACN model, which can be transformed from source code or a UML class diagram, and a change source. It outputs a list of files that may need to be changed together. The prediction algorithm works on the condensation graph generated from the ACN as we briefly introduced in Section 2. The condensation graph is first constructed based on the logical expression of the ACN; the edges are then trimmed according to the dominance relation that formalizes the concept of *design rules*, and the vertices are collapsed into strongly-connected components.

In the condensation graph, we assign a weight to each component to indicate its likelihood to be impacted by the change source. Variables with the highest weights at the end of the algorithm are recommended to the user. We first assign a weight to a component according to the number of sub-ACNs (identified using Cai and Sullivan's [7, 9] ACN decomposing algorithm) involving it. Since a sub-ACN represents all the decisions needed to accomplish a task, all the sub-ACNs that contain the change source will contain all the variables that are likely to be impacted by the change source. The more sub-ACNs a variable is involved in, the more likely the variable will be influenced and changed because of the higher probability that one of the many tasks represented by these sub-ACNs will be changed. By us-

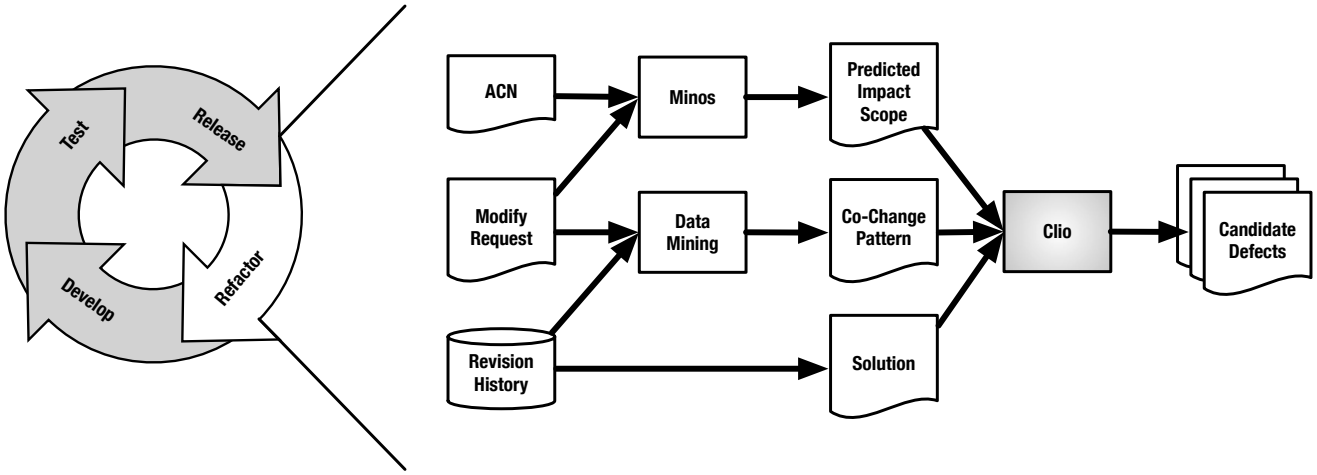

Figure 3: Approach Overview

ing sub-ACNs, we allow components that are semantically dependent, but not necessarily syntactically dependent, to potentially be recommended in the impact scope.

Next, we modify the weights by the ACN graph structure. By definition, an edge (u, v) in an constraint graph represents that a change in u potentially causes a change in v. So we look at all elements that may be affected by changing the change source, either directly or indirectly, by taking all the vertices reachable from the change source in the constraint graph. We modify the weight assignment based on the rationale that, in the paths starting from the source and outgoing to all the variables that the source influences, the closer a variable to the source, the more likely it will be impacted by the change source.

The rationale is as follows: according to the definition of the constraint graph, if two variables are directly connected, it means that they appear in the same logical expression, changing one will highly probably influence the other. The further two variables are connected in the graph, the lower the probability that one will influence the other because they may be indirectly related in multiple logical expressions. Thus we perform a breadth-first search (BFS) on the condensation graph, starting at the change source's interface variable. During the search, we record the distant of how far each variable is from the change source. Then we divide the initial weight of each variable by its distant.

Variables that are not visited by the traversal have their weights divided by a constant, larger than the longest distance. We empirically found, through experimentation on modification requests in the first release of Hadoop, that dividing by the total number of variables was an effective value. An unvisited variable can influence something that the change source also influences but there are no direct nor indirect dependencies between it and the change source. Instead of simply discarding these unvisited variables, we apply a penalty to lower their probability to be recommended. As a result, the design rules that dominate the change source are less likely to be predicted because of the penalty.

In the last step of calculating the recommendation weights, for all variables in the same Java package[3] as the change source, we increase the weight by a multiplicative factor. Our experiments on the first version of Hadoop find that using the number of sub-ACNs is effective. We increase the weight of these variables because classes in the same package generally have high cohesion and therefore tend to change together.

After calculating these weights, we take the ten elements with the highest weights and recommend those to the user as part of the impact scope. Since the algorithm work on the constraint graph trimmed according to the design rules in the system, that there is a penalty if changes to design rules are due to changes to its subordinating decisions, we call this algorithm a *design-rule based* prediction.

4. EVALUATION

Ideally, we should apply our approach to the life cycle of a real, ongoing project and observe its effectiveness in identifying design defects. To preliminarily validate our approach, before applying to a real project, we investigate the version histories of two open source projects, Hadoop Common and Derby, in a retrospective way. For each system, we look at multiple releases and the modification requests along with their solutions recorded by their bug tracking systems. We generate an ACN for each version from the source code, predict impact scope from it for each modification request completed before the next release, calculate co-change pattern, and identify potential design defects. After that, we examine the modification requests and code base to determine whether the potential defects are real design defects.

The design-rule-based impact prediction algorithm was implemented as a component of our ACN modeling and analysis tool, Minos [24, 29]. The co-change pattern algorithm was implemented as a separate utility program. We also implemented a tool, called *Clio*, to automate the process of calculating candidate defects. We ran our experiments on a 2.53Ghz Intel Core 2 Duo MacBook Pro with 4GB of RAM. Running our prediction algorithm on the test subjects took about one minute per version. Running the co-change pat-

[3]Since our case studies are implemented in Java, we used Java packages but other object-oriented languages have similar constructs (e.g. C++ namespaces) that can be used instead.

tern algorithm took about 8 minutes for the last version of Hadoop, which involves all available transactions in the version history. The implementations of our algorithms are in prototype states so performance can still be greatly improved. In the next subsections, we describe our evaluation subjects, procedure, and results.

4.1 Subjects

In this subsection, we describe our evaluation subjects, Hadoop Common and Derby. We chose these two systems first because they both employ an effective bug tracking system, JIRA[4], so that we can easily identify modification requests and their solutions. Because we have to examine the source code to verify if a potential defect is really a defect, their relatively short version histories make this job easier.

Moreover, both systems appear to be well-modularized. We assessed these systems using the *independence level* (IL) metric proposed in our recent work [25]. Given a design clustered into a *design rule hierarchy* (DRH) [25, 30], this metric measures the portion of the system that consists of truly independent modules (e.g. modules which can be substituted without affecting the rest of the system). The fewer layers in the DRH and the larger portion of the system that falls into the last layer (indicated by a higher IL value), the better the system is modularized.

The Hadoop DRH shows that, over the 15 versions we studied, the number of layers only increases from 6 to 9, and the IL increases from 48% to 49%; for the 9 versions of Derby, only one layer is added (11 to 12) with a slightly decreased IL (from 52.5% to 51.4%). Since Derby is 5 times bigger than Hadoop, it appears to be even better modularized. The fact that both systems are well-modularized and well-maintained means that the number of potential design defects should not be large, if any.

Hadoop Common: Hadoop is an open source map/reduce system for distributed computing, written in the Java programming language. Since Hadoop consists of several subprojects, we focused on the *Hadoop Common* (formerly called *Hadoop Core*) sub-project for our evaluation. Of the 20 versions of Hadoop available, we focused the first 15 releases, versions 0.1 to 0.15, covering a time period of about three years of development. We used a total of 275 modification requests as inputs to our defect detection algorithm.

Derby: Derby is a relation database system, implemented in the Java programming language. It was originally a commercial product, developed by IBM and released to the open source community—becoming an Apache project in late 2004. For our evaluation, we looked at 9 releases of Derby. The versions in chronological order of the release were 10.1.1.0, 10.1.2.1, 10.1.3.1, 10.2.1.6, 10.2.2.0, 10.4.1.3, 10.3.3.0, 10.4.2.0, and 10.5.1.1 (notice version 10.4.1.3 was released prior to 10.3.3.0). The versions we examined covered almost four years of development, and included 161 modification requests that had solutions. Although Derby also employs JIRA for maintaining modification requests, many of the modification requests do not have recorded solutions. Hence, fewer modification requests were available for use in our evaluation.

[4]http://www.atlassian.com/software/jira/

4.2 Evaluation Procedures

For each release, we used JIRA to collect the set of modification tasks that were completed during that release's development. Because JIRA records the change sets that close each modification request, we can also easily obtain the solution of each modification task. We manually examined the task descriptions to identify the change source. For example, modification request #174 of Hadoop states: "The launching application (via JobClient) checks the status once a second. If any timeouts or errors occur, the user's job is killed." This request is to modify the system to be more persistent when checking on the status of a running job. From this description, we identify the `JobClient` class as the change source. In our evaluation, we specify only one file as the change source for each modification task, based on its description. Our future work is to extend our approach to consider multiple change sources.

We reverse engineered a UML class diagram for each release and converted the diagram into an ACN, using the approach presented in our previous work [20, 29]. Then we used the ACN, along with the modification task's change source, as input to Minos for predicting change impact. Using the same change source, we ran the co-change pattern algorithm, and performed the impact comparisons with actual solutions. Our tool, Clio, recorded all candidate defects identified, as well as the support and confidence heuristic values.

4.3 Results

In this section, we describe the support and confidence values we empirically chose for each subject system (in order to avoid creating excessive false positives), and the design defects identified by our technique.

Hadoop.

We investigated four potential defects in Hadoop, revealed by our approach using the following heuristic criteria: a candidate defect must have support no less than 3 and confidence no less than 0.75, for at least two releases of the software. We defined these heuristic threshold values empirically, by examining the produced candidate defects. Although choosing a high minimum confidence level may hide some actual design defects (false negatives), we find this confidence level to be a good balance between looking at excessive numbers of false positives and identifying important design defects.

Based on these threshold values, out of the four potential defects we examined, only one was a false positive. Additionally, two of the potential defects were confirmed as true defects by the fact that these parts were refactored in later versions of the code. Although we looked at several other candidate defects in the system that are unlikely to be design defects (false positives), we do not report them here because they did not meet the heuristic threshold we set. Below, we specifically describe each of the four potential defects that did meet the heuristic threshold.

DistributedFileSystem: As described in Section 3, modification request #51, in version 0.1, describes changing the `DistributedFileSystem` class while its parent class `FileSystem` is impacted and another child of the `FileSystem`, (`LocalFileSystem`) is also contained its solution.

Our approach identified these classes as a candidate defect. In fact, the `FileSystem` class is changed 47% of the time `DistributedFileSystem` is changed and the `LocalFileSystem` class is changed 37% of the time `DistributedFileSystem` is changed, yet there are no ACN dependencies between `DistributedFileSystem` and `LocalFileSystem`. Figure 1 shows the UML relationship between these three classes. By release 0.3, our tool reported that `DistributedFileSystem` and `FileSystem` are a potential design defect with support of 3 and confidence of 1.

Upon further investigation of the source code for the three classes, we found an unusual design. The `FileSystem` class contained methods to construct both the `LocalFileSystem` and `DistributedFileSystem` classes. From a design rule perspective, the parent class `FileSystem` is a very unstable design rule and poor design because changes to its child require changes to itself and its other children.

Our intuition that this is a problematic issue was confirmed when we looked forward through the revision history and found that by release 0.19, the method to construct `DistributedFileSystem` had been deprecated in `FileSystem`, in favor of a method in a external class. It appears that the design was refactored, due to the tight coupling of the unstable design rule.

ReduceTask: Modification request #1127 in version 0.12 is titled "Speculative execution and output of Reduce tasks" and it describes a change to the `ReduceTask` class. The candidate defect for this modification request includes the `Task` class, which is the parent class of the `ReduceTask` class. In fact, the `Task` class is one of the classes most often changed with the `ReduceTask` class: by release 0.14, the `Task` class is changed in the same transaction as the `ReduceTask` class nearly 40% of the time. If the parent class is changing so often when the child class changes, it indicates a possible need to refactor or redesign. As early as version 0.6, our approach revealed this tight coupling between these two components with a support of 3 and confidence of 0.5.

To verify this suspicion, we looked forward in the version history and saw, in fact, a refactoring occurs in version 0.19. Modification request #3150 "defines a new interface for committing output files" and removes most of the changes made to the `Task` class from modification task #1127. Instead, a new `OutputCommitter` interface and several implementing classes are added as part of a refactored design.

JobTracker: Starting with version 0.2, we see that very often when the `JobTracker` changes, the `JobInProgress` and `TaskInProgress` also change, and they are identified in a candidate defect with support of 3 and confidence of 0.75. In version 0.3, the support increases to 4 and the confidence of 0.8. We examined the source code, and found that there is a very tight coupling between these components. In fact, there is a cyclical dependency—`JobInProgress` aggregates `JobTracker` and `TaskInProgress`; `JobTracker` constructs instances of `JobInProgress`, passing itself for the `JobInProgress` to later use. This cyclical dependency seems to create high coupling between these components and may hinder future maintenance.

Our approach also revealed code clones in `JobTracker` for using `TaskInProgress` and `JobInProgress`. For example, there is a `getMapTaskReports` method and a `getReduceTaskReports` method whose implementation only differ by variable names. Within these methods there are cloned forloops to interact with `TaskInProgress` objects. By version 0.19, these methods have been cloned into more methods. However, as of version 0.19, we have not found any documented attempts to refactor the design to confirm our suspicion of this design defect.

NameNode: Using the heuristic threshold values we described earlier, we found only one false positive design defect in Hadoop. Although starting with version 0.6, `NameNode` and `FSNamesystem` appear as a candidate defect with support of 8 and confidence of 0.8, we were unable to find any design defect involving the two. There may be, in fact, a design defect but since we are not designers of the project, we were unable to determine whether this potential defect is a true defect. Hence, we classify this result of our approach as a false positive.

Derby.

Since the number of modification requests we obtained from Derby was significantly less than for Hadoop, we were unable to use the same heuristic threshold values. The highest support value obtained by any candidate defect was only 3, and only reached in the last version we examined. Therefore, we relaxed the heuristic thresholds used to having either a support of no less than 3, or a support of 2 and a confidence of no less than 0.75, for at least two versions.

Although we identified three potential design defects in Derby, we were unable to confirm the defects with any modification requests or other documentation that indicate future refactoring. A possible explanation for this lack of evidence is that the defects we found were identified in very recent releases of Derby. For example, the `ResultSet` potential defect, did not meet our heuristic threshold criteria until version 10.5.1.1; however, only one version of the system has been released since then—barely four months since that release.

ResultSet: The `org.apache.derby.client.am.ResultSet` and `org.apache.derby.client.am.Statement` candidate defect reached a support of 2 by version 10.2.2.0 and a support of 3 by version 10.5.1.1. It was one of only two candidate defects to reach a support value of 3. However, even by this last release they have a confidence of only 0.375. Similar to the `JobTracker` candidate defect in Hadoop, the `ResultSet` and `Statement` are very tightly coupled. The `Statement` constructs the `ResultSet` and keeps a reference to the constructed object. However, the `ResultSet` implementation extensively uses the `Statement` class. Although we are not developers in the project, we believe this is a design defect and that the `ResultSet` should instead use the `java.sql.Statement` interface, so that the actual `Statement` implementation can change without affecting it. Since the `java.sql.Statement` interface is (a design rule) defined by the JDBC standard, it is more stable and less likely to change than the specific implementation. We believe this defect finding to be important because the `Statement` class has many children (e.g. `PreparedStatement`, `CallableStatement`) and this tight coupled is inherited by all of them.

EmbedResultSet: Along with the `ResultSet`, the `EmbedResultSet` and `EmbedStatement` is the only other candidate defect reaching a support value of 3. This candidate defect appears in version 10.4.2.0 with a support of 2 and confidence of 0.4, and after the release of version 10.5.1.1, it has a support of 3 and confidence of 0.5. Like the `ResultSet` candidate defect, the `EmbedResultSet` and `EmbedStatement` classes are cyclically dependent and highly coupled. Since

this case is very similar to the prior case, we do not further elaborate on the details.

UnionNode: In two modification requests (support of 2), with the `UnionNode` as the change source, the `SelectNode`, `ProjectRestrictNode`, and `PredicateList` classes are identified as candidate defects by version 10.2.1.6. Since the confidence level remains 0.67 through the next several versions, we investigate them as a potential design defect.

Upon investigation, we found that the `SelectNode` and `PredicateList` are tightly coupled and in a cyclical dependency. The `UnionNode` and `ProjectRestrictNode` are not part of this design defect but they both use both of the components in the defect, and gave us a starting point to identify the defect. Although we suspect this cyclical dependency to be a design defect, we have not found any documented attempts to refactor the design, in order to confirm our hypothesis.

Connection: With a support of 2 and confidence of 1, starting with version 10.4.2.0, we investigated the `Connection` and `NetConnection` components as being part of a design defect. We investigated these components but were unable to find any design defect involving the two. Due to the lack of modification requests available and the fact that this candidate defect only has a support of 2, we believe it may have just a coincidence that these classes were modified together. With a support of 2 and confidence of 1, it implies that there were two modification requests with the `Connection` component as the change source and the `NetConnection` was in both solutions. Hence, we classify this result of our approach as a false positive due to the relaxed heuristic threshold values.

5. DISCUSSION

The minimum support and confidence values to consider a candidate defect as a potential design defect was selected empirically in our evaluation. These values may be adjusted according to the size of the system, quantity of modification tasks available, and the length of the version history. As we saw in our evaluation, since Derby had significantly fewer modification tasks than Hadoop (due to quality of documentation), we needed to lower the minimum support and confidence levels. However, lowering these values for other systems may potentially increase the number of false positives generated by the approach. In addition, although the high minimum confidence value reduced the number of false positives we encountered, it may have also created false negatives (design defects caused by design rule violation that were not identified). Due to the large number of modification tasks we collected, and the fact that we lack domain knowledge on the system designs, we did not perform extensive analysis of false negatives for the subject systems. It is our future work to empirically study the tradeoffs of adjusting minimum support and confidence levels, both in other systems and more rigorously for Hadoop and Derby, considering both false positives and false negatives.

The accuracy of the predicted change impact scope may vary with the ACN model in use. For example, the ACN model we used in this paper were automatically generated from UML class diagrams derived from the source code. Although these ACN models reveal more dependencies than pure syntactical dependencies, it is still possible that important dependencies are missing, such as the dependency between a client and a server, which can be detected using logical dependencies or in high-level models, such as a UML component diagram or architectural description languages. The `NameNode` candidate defect we investigated in the evaluation may have been identified as a potential defect due to dependencies not captured from the code-generated ACN. A future work is to evaluate the effectiveness of our approach using high-level, architectural models in conjunction with source code, in generating ACNs.

One challenge of our evaluation is to get enough modification tasks with traceable solutions. The Hadoop development community very thoroughly documents their modification tasks and the transactions that close them (solutions). Although Derby uses the same bug reporting system (JIRA) as Hadoop, the solution to modification tasks were recorded much more sparely. In fact, one of the most challenging parts of our evaluation was selecting subject systems with a significant number of modification tasks that have identified solutions. Many open source projects that we examined do not keep track of the transactions that close modification requests. Our results, however, show that it is worthwhile to maintain thorough modification request data for their potential use of detecting design defects.

We hypothesize that one reason behind the fact that we only found a few design defects is that these systems are well modularized. Further exploring the relation between design defects and its independency level is our future work. We also plan to investigate a system that is known to be not well-modularized in order to see if our approach can detect more design defects.

Another possible reason that we only detected a few defects is because these systems are relatively young and the modularity is not decayed. A reason we choose these systems is because their short revision histories enable us to manually verify the potential defects. We hypothesize that the longer the system is maintained, the more defects can be found. In addition, due to the relative youth of these systems we had difficulty confirming some of the defects we identified. For example, as we stated in Section 4, many of the potential defects we identified in Derby are from very recent modification tasks; so the designers may not have had the chance to identify these defects yet. Working with designers of a system to validate the defects identified by our approach is a future work.

We do not claim that this approach will detect all the design defects possible. Instead, we claim that this approach will effectively detect design defects caused by design rule violations that are manifested in the maintenance stage of the software. Our modularity conformation checking method, Glusta [21], can also detect similar defects. However, Glusta requires the existence of a high-level design model that represent correct design rules, which is usually not available, especially in large, open source projects. Additionally, Glusta, like many conformance checking techniques, works on static models—for example, A and B may change together but do not have any syntactic dependencies—so it may miss detecting the violation of such logical dependencies. In contrast, our defect detection approach takes co-change pattern into account and detects potential design defects by examining the difference between the pattern-based and design-rule-based predictions.

6. RELATED WORK

The relation between software dependency structure and defects has been widely studied. Hutchens and Basili [19] and Selby and Basili [23] were among the first to assess a system's failure proneness using dependency structure. Based on Stevens et al.'s [26] definitions of coupling and cohesion, Hutchens and Basili defined metrics to cluster code elements into modules in order to minimize the size of modules containing development errors.

Selby and Basili extended those results, finding that modules with lower coupling are less likely to contain defects than those with higher coupling. Briand et al. [6] found similar results through empirical studies, and they report a correlation between Chidamber and Kemerer's [11] coupling measures and the failure proneness of classes. Specifically, they assert that the frequency of method invocations is the main driving factor of fault-proneness in systems.

Eaddy et al. [13] considered design modularity in terms of how well a design localizes concerns (i.e. functional requirements whose implementation span multiple software modules), and performed three studies on the relation between level of concern scattering and number of defects. They concluded that, independent of the number of lines of code to implement a concern, the more scattered it is across modules, the more likely it is to contain defects.

Unlike the above work that aims to measure the likelihood of defects, our approach is used to identify concrete defects.

Fowler [16] describe the concept of "bad smell" as a heuristic for identifying designs to be refactored. Example bad smells include code clone and feature envy (when a class excessively uses methods of another class). Garcia et al. [18] proposed several architectural level bad smells. Detection of some specific bad smells have been extensively researched. In particular, detection of code clones has been quite popular. For example, Tairas and Gray [27, 28] used suffix trees and abstract-syntax trees to automatically detect clones.

To automate the identification of bad smells, Moha et al. [22] presented a tool, Decor, and domain specific language (DSL) to automate the construction of design defect detection algorithms. For example, to detect spaghetti code, the authors describe (using the DSL) that potential defect locations contain long methods, methods with no parameters, no inheritance, use of global variables, and no polymorphism. Based on such a description, their framework generates an algorithm that can analyze Java code looking for such properties. However, certain bad smells may be difficult to properly identify using only syntactic information. For example, Al-Ekram et al. [1] describe how "accidental" clones often appear in source code due to certain component interaction protocols, which may cause Decor to report many false positives. Our approach is different in that we identify defects based on the assumption relation among design decisions.

Instead of just identifying a particular bad smell such as cloned code, our approach is general enough to identify multiple kinds of "bad smells" that appear to be deviating from the modular structure determined by design rules.

The relation between logical dependencies [17] and defects has also been recently studied. Eick et al. [14] showed that increases in logical dependencies served as a good indicator for code decay. Cataldo et al. [10] argue that the correlation between density of syntactic dependencies and failure proneness is limited, and that the density of logical dependencies serve as a much better indicator. Our definition of *logical dependency* is different from theirs. They define that two components are *logically dependent* if they changed together frequently in that past. However, as we have explained, two components may change together because of design defects. Our approach defines that two components are *logically dependent* if one of them makes an assumption upon the other.

7. CONCLUSION

To address the problem that design rule violations may cause severe modularity degradation and incur high maintenance costs, but usually are not detectable using traditional software testing techniques, in this paper, we presented an approach to detect design rule violations by discovering co-change patterns in version history and assessing the differences between the pattern and the change scope predicted by the ACN model of the design. We evaluated our approach by investigating the version histories of Hadoop and Derby. Our approach identified multiple potential design defects such as poorly designed inheritance hierarchies, tightly coupled cyclic dependencies, as well as cloned code. Some potential defects revealed by our approach were indeed refactored in later releases, verifying the effectiveness of our approach.

8. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants CCF-0916891 and DUE-0837665.

9. REFERENCES

[1] R. Al-Ekram, C. Kapser, R. C. Holt, and M. W. Godfrey. Cloning by accident: An empirical study of source code cloning across software systems. In *Proc. International Symposium on Empirical Software Engineering*, pages 376–385, Nov. 2005.

[2] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proc. 9th IEEE International Conference on Software Maintenance*, pages 292–301, Sept. 1993.

[3] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.

[4] S. A. Bohner. Software change impacts - an evolving perspective. In *Proc. 18th IEEE International Conference on Software Maintenance*, pages 263–272, Oct. 2002.

[5] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society, 1996.

[6] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In *Proc. 21st International Conference on Software Engineering*, pages 345–354, May 1999.

[7] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, Aug. 2006.

[8] Y. Cai and K. J. Sullivan. Simon: A tool for logical design space modeling and analysis. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 329–332, Nov. 2005.

[9] Y. Cai and K. J. Sullivan. Modularity analysis of logical design models. In *Proc. 21st IEEE/ACM*

International Conference on Automated Software Engineering, pages 91–102, Sept. 2006.

[10] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*. To Appear.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[12] C. R. B. de Souza. *On the Relationship between Software Dependencies and Coordination: Field Studies and Tool Support*. PhD thesis, University of California, Irvine, 2005.

[13] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

[14] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, Apr. 2002.

[15] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

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

[17] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. 14th IEEE International Conference on Software Maintenance*, pages 190–197, Nov. 1998.

[18] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Proc. 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, Mar. 2009.

[19] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, Aug. 1985.

[20] S. Huynh, Y. Cai, and W. Shen. Automatic transformation of UML models into analytical decision models. Technical Report DU-CS-08-01, Drexel University, Apr. 2008. https://www.cs.drexel.edu/content/uploads/Research/DU-CS-08-01.pdf.

[21] S. Huynh, Y. Cai, Y. Song, and K. Sullivan. Automatic modularity conformance checking. In *Proc. 30th International Conference on Software Engineering*, pages 411–420, May 2008.

[22] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. Duchien. A domain analysis to specify design defects and generate detection algorithms. In *Proc. 11th International Conference on Fundamental Approaches to Software Engineering*, pages 276–291, Mar. 2008.

[23] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, Feb. 1991.

[24] K. Sethi. From retrospect to prospect: Assessing modularity and stability from software architecture. Master's thesis, Drexel University, June 2009.

[25] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna. From retrospect to prospect: Assessing modularity and stability from software architecture. In *Proc. 8th Working IEEE/IFIP International Conference on Software Architecture*, Sept. 2009.

[26] W. P. Stevens, G. J. Meyers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[27] R. Tairas and J. G. Gray. Phoenix-based clone detection using suffix trees. In *Proc. 44th ACM Southeast Regional Conference*, pages 679–684, Mar. 2006.

[28] R. Tairas and J. G. Gray. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering*, 14(1):33–56, Feb. 2009.

[29] S. Wong and Y. Cai. Predicting change impact from logical models. In *Proc. 25th IEEE International Conference on Software Maintenance*, Sept. 2009.

[30] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2009.

[31] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, Sept. 2004.

[32] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. 26th International Conference on Software Engineering*, pages 563–572, May 2004.