

Predicting Change Impact from Logical Models

Sunny Wong and Yuanfang Cai
Department of Computer Science
Drexel University
Philadelphia, PA, USA
{sunny, yfcai}@cs.drexel.edu

Abstract

To improve the ability of predicting the impact scope of a given change, we present two approaches applicable to the maintenance of object-oriented software systems. Our first approach exclusively uses a logical model extracted from UML relations among classes, and our other, hybrid approach additionally considers information mined from version histories. Using the open source Hadoop system, we evaluate our approaches by comparing our impact predictions with predictions generated using existing data mining techniques, and with actual change sets obtained from bug reports. We show that both our approaches produce better predictions when the system is immature and the version history is not well-established, and our hybrid approach produces comparable results with data mining as the system evolves. Furthermore, we show how to integrate our approach with ex post change sets to identify issues that suggest potential redesign or refactoring.

1. Introduction

During software maintenance, changes are often introduced to fix bugs and accommodate new requirements. To estimate the effort required in making the change and to avoid introducing defects as side effects, it is important to understand the impact scope of such changes. As Bohner [4] points out, estimating change impact based on both direct and indirect structural dependencies often overestimates the impact scope.

A recent direction is to predict change impact from historical data using data mining techniques [16, 17]. Such approaches provide recommendations based on how frequent two parts of the system have changed together and show promise in detecting dependencies that structure-based approaches miss. The problem is that the accuracy of these approaches rely on the existence of well-established version histories. Their effectiveness therefore degrades when the

project is relatively new or the design is refactored [16].

In this paper, we present change impact prediction approaches that are based on an extended software structural relation (not direct or transitive syntactical relations) so that these approaches can both provide useful recommendations and be applied to relatively young systems with limited version histories. One of these approaches also integrates data mining techniques to leverage knowledge obtained from version histories.

The approaches we present in Section 3 are based on the *assumption* relation among design decisions of a software project, represented by a logical framework called the *augmented constraint network* (ACN) [7, 8]. An ACN can express assumptions among classes, modules, aspects, as well as environmental conditions (e.g. memory size, requirements [7, 8, 13]). As the first paper to explore the change impact predicting ability of ACN modeling, we only consider the assumption relation among classes in object-oriented software—the assumptions that can be automatically derived from UML class diagrams. In Section 3.1, we describe how to translate class diagrams into ACNs.

We observe that the assumption relation derived from a UML class diagram reveals more dependencies than syntactic analysis but fewer dependencies than the transitive closure. For example, using the technique we present in Section 3.1, we identify 36 dependencies from the maze game system described in Section 2. While using Lattix [15], a static program analysis tool, only 24 dependencies are detected. Using ACN modeling, we also take the dominance relations among design decisions into consideration; that is, some decisions (such as an abstract interface implemented by multiple classes) theoretically, should not be affected by subordinate classes.

We are aware that it is possible that neither syntactical nor assumption relations can cover all the reasons that make two components change together. It is also not uncommon that theoretically stable decisions appear to be volatile in reality. These relations may be manifested easier by data mining, or other more complex techniques. As introduced

in Section 3.3, we complement the purely ACN-based prediction approach with the knowledge obtained using existing data mining techniques.

This paper also explores the possibility of using the purely ACN-based approach to identify potential design problems that need redesign or refactoring. The rationale is as follows. When making high level design 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. If the system is frequently changed in ways differing from these assumptions, then there could be a mismatch between the designer’s assumptions and reality, which may require improvement to the design.

For example, if changes to class *B* often requires changes to its parent class *A* then the design is not very stable, since 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 [3]; it should not be affected by its subordinate design decisions (its sub-classes). Strictly following design rule theory, it will not be predicted in the impact scope of *B*. If our ACN-based approach does not predict *A* to be in the impact scope of *B* but they frequently change together, then this may indicate a possible problem with the design.

We evaluate our approaches (Section 4) on 14 versions of the open source Hadoop¹ system using class diagrams that we obtain through reverse engineering of the code base. Using over 300 modification tasks as real maintenance scenarios, we compare the predictions of both our approaches with the actual change sets completed by developers, and with an existing data mining approach. We show that both our approaches produce better predictions when the system is immature with short version history, and our hybrid approach produces comparable results with data mining as the system evolves. In addition, we show that by investigating deviations between the purely ACN-based predictions and the actual change sets, we can potentially identify issues that suggest redesign or refactoring.

2. Background

We use a small example so illustrate the background of our work. In particular, we describe the logic model that we use to formalize the UML relations and develop the impact scope prediction approach, the *augmented constraint network* (ACN) [7, 8].

Figure 1 shows a UML class diagram of a small system for building a maze in a computer game, applying abstract factory pattern, as described in Gamma et al. [10]. A maze is defined as a set of rooms; a room knows its neighbors, such as another room, a wall, or a door to another room. The class `MapSite` is the common abstract

class for all the components of the maze. The UML class diagram shows two variations of the maze game supported by the abstract factory pattern: an enchanted maze game (`EnchantedMazeFactory`) with doors that can only be opened and locked with a spell (`DoorNeedingSpell`) and rooms that have a magic key (`EnchantedRoom`); and a bombed maze game (`BombedMazeFactory`) that contains rooms with build-in bombs (`RoomWithABomb`) and walls that can be damaged if a bomb goes off (`BombedWall`).

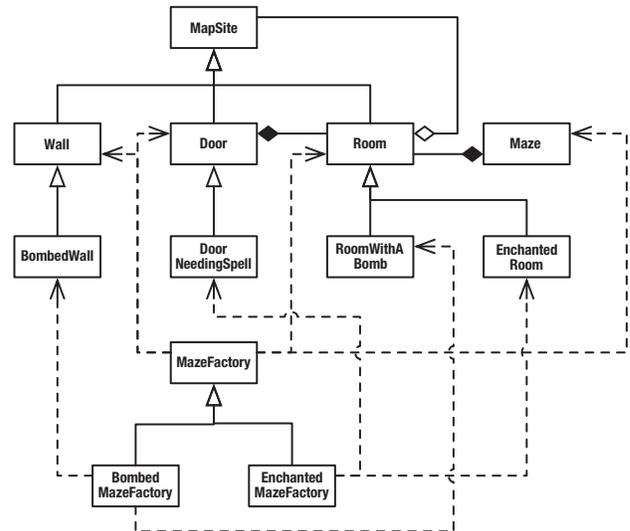


Figure 1. Maze Game UML Class Diagram

The UML diagram implies a number of assumptions and constraints among these classes. For example, the `MapSite` abstract class is assumed to be stable and not likely to change because changing it may impact its multiple sub-classes. The same is true for `MazeFactory`, which decouples the client from the concrete types of maze games. In Baldwin and Clark’s modularity theory [3], these classes are *design rules*, defined as stable design decisions that decouple otherwise coupled modules and create independent modules. Ideally, these design rules should not be impacted by changes to decisions that are subordinate to them. For example, `MapSite` is a design rule to `Door`, so changes to the class `Door` should not force the change to `MapSite`, otherwise other sub-classes may be affected.

Our recent work 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. An ACN consists of a *constraint network* that models design decisions and their relations, a *dominance relation* that formalizes the design rules, and a *cluster set* in which each cluster represents a different way to partition a design (not used in this paper).

A constraint network consists of a set of design *variables*, which model design dimensions (or relevant envi-

¹<http://hadoop.apache.org/core/>

```

1. MapSite_interface : {orig, other};
2. MapSite_impl : {orig, other};
3. Room_interface : {orig, other};
4. Room_impl : {orig, other};
5. Maze_interface : {orig, other};
6. Maze_impl : {orig, other};
7. Room_impl = orig => MapSite_interface = orig;
8. Room_impl = orig => MapSite_impl = orig;
9. Maze_impl = orig => Room_interface = orig;
10. (MapSite_impl, MapSite_interface);
11. (Room_impl, Room_interface);
12. (Room_interface, MapSite_interface);

```

Figure 2. Maze Game Partial Augmented Constraint Network

ronment 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* relations among design variables. In Figure 2, lines 1–6 are some variables of the maze game design, and lines 7–9 are some sample constraints. For example, line 7 models that the implementation of the `Room` class assumes that the interface of the `MapSite` 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 10–12. For example, line 11 indicates that the decision for how to implement the `Room` class cannot influence the design of its interface; in other words, we cannot arbitrarily change the `Room` class’s public interface to simplify the class’s implementation because other components may rely on that interface.

As we show in the next section, to assess change impact, we consider the system’s subsystems (features, functions), and assess how many systems a variable is involved in. The approach is based on Cai et al.’s [8] prior work of decomposing an ACN model into sub-ACNs to increase the performance of constraint solving. The basic idea is to model the constraint network as a directed graph. In this graph, each vertex represents a design variable. Two variables are connected if and only if they appear in the same constraint expression. Then the edges of the directed graph are removed according to the dominance relation of the ACN: if A cannot influence B , then the edge from A to B is removed from the graph. We then compute the condensation graph (graph of strongly-connected components) of this graph.

Figure 3 shows a partial maze game condensation graph generated from the maze game ACN. Note that the edge directions in the graph may seem counter-intuitive. This is because the edges do not represent the direction of dependence but rather the direction of possible influence. In other words, if an edge (u, v) exists in the graph then a change in the decision for variable u may potentially influence the decision on variable v .

To generate sub-ACNs, we put all the variables along the paths ending with the same minimal elements into a sub-ACN with the relevant subset of constraints, dominance relation and cluster set. As a result, the ACN is decomposed into a set of sub-ACNs. We observe 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. For example, Figure 3 shows that one of the sub-ACNs will contain the variables `EnchantedMazeFactory_impl`, `EnchantedMazeFactory_interface`, `MazeFactory_interface`, `EnchantedRoom_interface`, `Room_interface`, and `MapSite_interface`. This sub-ACN contains all the decisions needed to implement the `EnchantedMazeFactory_impl` feature.

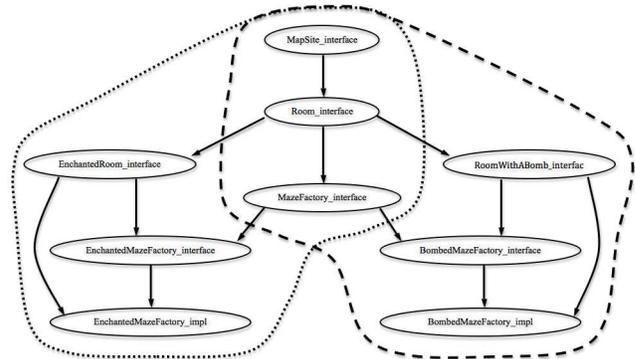


Figure 3. Partial Maze Game Condensation Graph

3. Prediction Approaches

In this section, we present two approaches to predict the impact scope of a given change based upon the assumption relation among classes derived from UML class diagrams. In Section 3.1, we first present how to extract and formalize logical assumption relations implied by UML class diagrams into ACN. In Section 3.2, we present a change impact prediction approach purely based on the derived ACN. In Section 3.3, we present a hybrid prediction approach that adjust the purely ACN-based prediction with knowledge obtained from version history. As the input to our predicting approaches, a UML model can be either constructed by the designer or reverse engineered from source code.

3.1. UML Translation

In this section, we present the formalization of UML class diagrams using augmented constraint networks. Our approach currently supports translating of classes, interfaces, and the major binary relations of class diagrams: generalization, realization, dependency, aggregation, composi-

tion, directed association, and owned element. For the sake of space, we only present the formalization of UML classes, interfaces, and one relation. Our previous work [12] details the formalization of all these relations. These UML relations are all supported by IBM Rational Rose² and used in Hadoop, the real system we experimented with.

Class and Interface From a decision-making perspective, each class consists of two design dimensions: an interface dimension, and an implementation dimension. We thus model a class, *A*, using two variables: *A_interface* and *A_impl*. Each dimension can vary, so we model the domain of each variable with at least two values {*orig*, *other*}, where *orig* models the current decision and *other* models an unelaborated new choice for the decision. Our interface variable is different from the *interface* construct in Java and other object-oriented languages. We generally view all the non-private part of a class as an interface that can be used by other classes, and view all private part of the class as the implementation detail hidden from other classes.

The ACN translated from class *A* is shown as below. Besides the two variables, we use a logical expression to model that the implementation of *A* makes assumptions about its interface. We also assume that the interface of a class dominates its implementation, which is translated into a dominance relation. As a result, we translate the class *A* into the following ACN:

```
Constraint Network:
  A_interface : {orig, other}
  A_impl : {orig, other}
  A_impl = orig => A_interface = orig
Dominance Relation:
  (A_impl, A_interface)
```

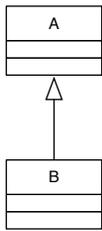
Since a UML interface, *I*, has no implementation, we model it using only one variable, the interface variable:

```
I_interface : {orig, other}
```

Generalization Table 1 shows a *generalization* relation depicted in UML, in which *A* is the general element and *B* is the specific element. Since *B* inherits the features of *A*, this means the design decision of *A*'s interface dominates and influences *B*, both its interface and its implementation, which is modeled using the first two logical expressions shown in the table.

Moreover, the implementation decision of *A* also influences the implementation of *B*. Although the changes in *A* will be propagated to *B* implicitly, and the developer may not need to change *B*'s code, s/he should be aware of this change to avoid unwanted side effects. The third logical expression makes this relation explicit. The *A_impl* also dominates *B_impl* because a change in *B* should not force

Table 1. Generalization

UML	Augmented Constraint Network
	<p>Constraint Network:</p> <pre>B_impl = orig => A_interface = orig B_interface = orig => A_interface = orig B_impl = orig => A_impl = orig</pre> <p>Dominance Relation:</p> <pre>(B_impl, A_interface) (B_impl, A_impl) (B_interface, A_interface)</pre>

changes in *A*, which may be inherited by other classes. The dominance relation is thus constructed accordingly.

3.2. Logic-based Prediction

Our purely ACN-based change impact prediction approach starts from an ACN, derived from a UML class diagram, and at least one class that initiates the modification task, the *change source*. The approach outputs a list of files that may need to be changed together. The prediction algorithm works on the constraint graph generated from the ACN. The concrete steps are as follows.

The first step is to figure out all the the classes that are logically related to the change source. Figure 3 depicts a partial constraint graph generated from an ACN model of the maze game example. Considering *Room* as the change source, all the edges going out of *Room_interface* are the classes that will be potentially impacted by the change source; and the edge coming in *Room_interface* links to a class that may influence the source. To simplify our explanation, we consider this partial system as the full system. Instead of recommending all these files as equally likely to be impacted the source, we rank the likelihood to be impacted of each class by assigning weights to them. Variables with the highest weights at the end of the algorithm will be recommended to the user.

We first assign a weight to a variable according to the number of sub-ACNs (as identified by Cai et al.'s [7, 8] 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, and the more sub-ACNs a variable is involved in, the more likely the variable will be influenced and changed.

For example, Figure 3 depicts the condensation graph for part of the maze game. We differentiate the two sub-ACNs, representing subsystems of *enchanted* and *bombed* maze respectively, with different lines styles. Among these variables, the *MazeFactory_interface* variable is involved in both sub-ACNs, meaning that it is more likely to be

²<http://www-306.ibm.com/Soft/awdtools/developer/rose/>

changed since the change in `Room` will impact one or both subsystems. We thus assign an initial weight to each variable based on how many of the relevant sub-ACNs it appears in: `MazeFactory_interface` has a weight of 2 while `EnchantedRoom_interface` has a weight of 1.

We further 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 source. 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.

Optionally, we can further adjust the weight to accommodate the fact that subordinating decisions do occasionally affect design rules. In an ACN constraint graph, the edges that go from a subordinate decision to a design rule are all removed, reflecting the rationale that design rules should not be affected by subordinate decisions. However, in reality, some classes turn out to be not as stable as the designer wishes.

To accommodate this reality and improve the accuracy of prediction, we perform a BFS again on the change source's interface variable, but this time following incoming edges to visit all variables that can influence the change source. Therefore, allowing design rules to be recommended. We augment the distance value with a constant penalty to indicate that changing design rules is not as common as changing non-design rules. Our experiments use a penalty of one.

To assess how much the reality deviates from the designer's intentions, we can skip the second BFS, generate predictions based on strong assumptions that design rules should never be changed by subordinate decision, and compare with the actual solutions in bug reports. In Section 4, we discuss how a designer can identify potential design problems by examining these deviations.

Variables that are not visited by either traversals have their weights divided by a constant, larger than the longest distance. In our experiments, we found 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. We do not simply discard these unvisited variables because it is possible that the unvisited variable can be affected by the variables that it influences.

In the last step of calculating the recommendation weights, for all variables in the same Java package³ as the

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

change source, we increase the weight by a multiplicative factor. Our experiments use the number of modules for this value. 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. The next subsection describes how we use an available version history information to apply a post-processing to the weights before selecting the recommendations for our hybrid algorithm.

3.3. Hybrid Prediction

Even though a UML-based logical model extracts more dependencies than syntactic analysis, it is possible that there are some dependencies that cannot be picked up by either approaches. It is also possible that some variables change together more often than others even though they have similar probability to change based on their ACN relations. This information may be reflected in the version history. Thus, we propose a hybrid algorithm that feeds the purely ACN-based approach, described above, with knowledge obtained from version history.

The idea is to mine the version history to find the percentage of how often a variable changes concurrently with the change source. Zimmermann et al. [17] refer to this percentage as the *confidence* level. We use this percentage to apply a post-processing on the weights of the purely ACN-based algorithm. After finding weights with the ACN-based algorithm, we multiply the weight of each variable by the percentage from data mining. For example, if there are 50 transactions with changes to *A* in the version history and out of those 50, *B* appears 10 times while *C* appears all 50 times, then we multiply the weight of *B* by 0.2 and the weight of *C* by 1.

4. Preliminary Evaluation

To assess the prediction quality of our approaches, we applied them to an open source software project, and compared the quality with that of an existing data mining technique. Our hypotheses are:

1. When the version history is not well-established, both of our approaches produce better predictions than the data mining approach.
2. When the version history matures, our approaches produce comparable results with that of data mining.
3. The hybrid approach produces better results than the purely ACN-based approach.
4. The deviation of the actual change set from the purely ACN-based approach indicates potential design issues that may require redesign or refactoring.

We choose Hadoop as our evaluation subject. Hadoop is an open source map/reduce system for distributed computing, written in the Java programming language. We choose this project because it is a relatively young project with only three years of development history and no major releases. Additionally, it employs an effective bug tracking system, JIRA⁴, that allows us to identify the solution change sets to assess our predictions.

4.1 Terms and Metrics

We use the following terms throughout this section, consistent with the definitions of Ying et al. [16]. A *modification task* can be either a bug fix or enhancement task. A *solution* is the set 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 checked in together.

In Table 2, we present some basic data regarding to the releases of Hadoop that we studied. Comparing with the systems studied by previous data mining work [16, 17], Hadoop has relatively few transactions. Its scale allow us to manually identify change sources and examine the reasons behind discrepancies of prediction and reality.

Table 2. Version Statistics

Version	# Transactions	# Modification Tasks
0.1.0	136	17
0.2.0	76	12
0.3.0	48	13
0.4.0	52	18
0.5.0	68	15
0.6.0	52	14
0.7.0	49	15
0.8.0	50	10
0.9.0	54	18
0.10.0	69	21
0.11.0	75	22
0.12.0	79	46
0.13.0	242	49
0.14.0	377	39
<i>Total</i>	1427	309

We use the standard metrics of *precision*, *recall*, and F_1 to measure the quality of predictions. Precision measures how many recommended files are actually involved in a solution. Recall measures how many files involved in solutions are recommended by the prediction approach. Precision and recall are complementary metrics. A high-precision prediction may recommend a small number of files that are all involved in the solution, but many other files

involved in the solution may not be recommended (low recall). A high-recall prediction may recommend all the files in a solution, but may also recommend many files that are not relevant (low precision). The values of both should be maximized in a balanced way (F_1) so that the user can find relative files without looking at large numbers of recommended files (precision), and that most files involved in the actual solutions can be predicted (recall). Mathematically, given a change source f_S and solution f_{sol} for a modification task m , we define:

$$\begin{aligned}
 \text{precis}(m, f_S) &= \frac{|\text{correct}(m, f_S)|}{|\text{recomm}(f_S)|} \\
 \text{recall}(m, f_S) &= \frac{|\text{correct}(m, f_S)|}{|f_{sol} - f_S|} \\
 F_1(m, f_S) &= \frac{2 \cdot \text{precis}(m, f_S) \cdot \text{recall}(m, f_S)}{\text{precis}(m, f_S) + \text{recall}(m, f_S)}
 \end{aligned}$$

4.2. Evaluation Procedure

For each release, we used JIRA to collect the set of modification tasks that are completed prior to the next release. Because JIRA records the change sets that close each modification request, we can also easily obtain the solution of each modification task. We assume for this evaluation that the solutions correctly implement the changes, and accurately show the actual change impact. For each modification task, we manually examine the task description to identify the change source. For example, modification request #174 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 between one and three classes as the change sources for each modification task, based on its description. Most of the modification tasks were initiated from only one class.

To apply the purely ACN-based and hybrid prediction approaches, we first reverse engineered the code base of all 14 releases into 14 UML class diagrams, and then translated the UML models into 14 ACNs according to the formalizations presented in Section 3.1. We make the first set of predictions using the purely ACN-based approach. Additionally feeding the first approach with knowledge obtained from the version history, we make another set of predictions using the hybrid approach. A third set of predictions is generated using a data mining approach following Zimmermann et al.’s work [17]. The basic ideas are as follows: given a change source element, the data mining approach analyzes the revision history for all transactions containing that element, keeping track of other files that appear in the same transactions. Given a pair of elements x_1, x_2 , the *con-*

⁴<http://www.atlassian.com/software/jira/>

fidence that x_2 is impacted by a change in x_1 is defined as the number of transactions that contain both x_1 and x_2 , divided by the number of transactions that only x_1 . Files are recommended based on their confidence level.

Following Zimmermann et al.’s work [17], we only report the top ten files recommended by each approach, because the user’s willingness to work through the recommended file list decreases dramatically as the length of the list increases. For modification tasks with more than one change source, we obtain the top ten recommendations for each class and take the union of these sets of recommendations as the final recommendation.

The next subsection describes our evaluation results. We ran our experiments on a 2.16 GHz Intel MacBook Pro with 3GB of RAM. With version 0.1.0 of Hadoop, it took 4 seconds to convert the UML diagram into an ACN. On average, for each change source class, it took 2 seconds to run the purely ACN-based prediction algorithm and 3 seconds to run the hybrid algorithm. As the system grew larger, with version 0.14.0, it took 16 seconds to convert the UML. It took 16 seconds to run the purely ACN-based prediction algorithm and 32 seconds to run the hybrid algorithm. The implementations of our algorithms are in prototype state so performance can still be greatly improved.

4.3. Evaluation Results

As an example, Table 3 shows the prediction results for all the modification tasks in version 0.6.0, using the hybrid approach. The first column lists the modification task number; the second column lists the actual number of files in the solution (excluding the change sources); the third column lists the number of files our hybrid approach correctly predicted; and the fourth column lists the number of files our approach recommended. The table also shows the average precision and recall values of the hybrid approach for this version.

Table 3. Example Modification Task Results

Task	Actual Impact	# Correct	# Recommended
560	6	4	10
559	1	1	18
550	1	1	10
527	7	5	10
522	3	1	18
519	5	3	10
513	3	0	10
438	2	0	10
423	1	0	10
306	9	4	10
293	1	1	10
288	5	2	10
263	13	5	10
243	5	3	10

Precision: 20.27% Recall: 48.39%

Table 4 reports the average value of each predictability metric for each release, in percentages. For each metric, the data mining values are placed in the middle to ease the comparison with the purely ACN-based and hybrid approaches. The Δ ACN and Δ Hybrid columns show how much better (or worse, if negative) our approaches performed than the data mining approach. Shaded cells indicate when one of our approaches produces better recommendations than the data mining approach.

Although the recall and precision measures seem to be low for all three approaches, as Ying et al. [16] point out, the value of an approach lies in the ability to provide helpful recommendations. Table 3 shows that in version 0.6.0, the hybrid approach is able to predict most of the files that are actually impacted in the first 10 files recommended. For example, for modification task 527 whose solution has 7 files, of the 10 recommended files, 5 are actually involved in the solution. The value of these prediction approaches become clearer considering the fact that a file may have dozens of dependents that can be impacted. Since the approaches only recommend ten files, this is already much fewer than examining all dependents. Next, we test the hypotheses proposed at the beginning of this section against these results.

Table 5. Solution Statistics

Cardinality of Solutions	Count
2	94
3	53
4	36
5	30
> 5	96
<i>Total</i>	309

Prediction ability comparison for the first set of releases. Comparing purely ACN-based versus data mining prediction, we observe that of all the 14 versions, although the purely ACN-based approach outperforms the data mining approach only 6 versions in precision, 8 versions in recall, and 7 versions in F_1 , all its victories are within the first 11 versions. The implication is that in comparison to the traditional data mining approach, the purely ACN-based approach performs better when the version history is short and the system is relatively new. The hybrid approach outperforms the data mining approach more times in the first 12 versions than the pure ACN-based approach does. The result shows that the first hypothesis holds.

Prediction ability comparison when the system matures. In versions 13 and 14, the data mining approach outperforms both ACN-based and hybrid approaches. However, the differences between the data mining results and hybrid results are not significant: the majority of differences are around only one percent, showing that the hybrid approach produces comparable results with that of data mining. Although we hypothesize that both approaches would be comparable with the data mining approach as the system

Table 4. Evaluation Results

Version	Precision			Recall			F1		
	Δ ACN	Mining	Δ Hybrid	Δ ACN	Mining	Δ Hybrid	Δ ACN	Mining	Δ Hybrid
0.1.0	5.34	16.3	6.66	10.9	32.7	12.9	7.17	21.8	8.78
0.2.0	4.70	25.7	4.70	9.62	36.5	9.62	6.48	30.2	6.48
0.3.0	-0.50	15.9	-0.50	-1.67	36.7	-1.67	-0.79	22.2	-0.79
0.4.0	-0.20	19.9	0.94	1.25	46.3	3.75	0.02	27.8	1.59
0.5.0	4.44	15.4	5.58	14.3	41.1	17.9	6.81	22.4	8.55
0.6.0	2.60	17.5	2.74	4.84	43.5	4.84	3.44	25.0	3.57
0.7.0	-1.50	14.6	-0.84	-4.55	50.0	-2.27	-2.26	22.6	-1.24
0.8.0	4.00	9.00	5.00	20.0	45.0	25.0	6.67	15.0	8.33
0.9.0	2.09	20.4	1.90	4.00	37.0	3.00	2.74	26.3	2.34
0.10.0	-4.19	25.1	-3.26	-6.67	56.7	-4.44	-5.30	34.8	-3.99
0.11.0	-3.09	29.1	1.76	1.37	45.9	8.90	-2.06	35.6	3.87
0.12.0	-1.05	15.1	0.33	-0.40	21.3	-0.25	-0.87	17.7	0.14
0.13.0	-1.69	18.1	-0.36	-1.42	25.4	-1.27	-1.65	21.2	-0.69
0.14.0	-4.71	24.8	-1.42	-1.23	32.2	-0.93	-3.65	28.0	-1.26

matures, the results conclude that only the hybrid approach satisfies this hypothesis. Since the purely ACN-based approach places a penalty to recommend design rules in the impact, modification of design rules in reality may be the reason for the purely ACN-based approach’s performance degradation. We further explore this deviation from reality in the next subsection. A future work remains to continue this study for newer versions of Hadoop and see if the hybrid approach remains comparable with data mining.

Pure logic-based approach vs. hybrid approach. The hybrid approach outperforms the purely ACN-based approach according to all the metrics in all versions. Considering the differences between these two approaches, this result is not surprising: two files may be assigned the same weight solely based their positions in the constraint graph, but if one file changes more frequently than the other, the hybrid approach would leverage this additional version history information and produce more accurate predictions.

In the next subsection, we discuss our fourth hypothesis, regarding the use of our prediction algorithm to identify design issues.

4.4. Identifying Design Issues

The ACN derived from a UML class diagram implies a hierarchy, modeling that subordinate classes should not influence design rule classes. For example, a parent class is a design rule to its sub-classes because changing the parent due to the change in one child may cause unwanted side effected in other children. It is possible that although the designer intends for these design rules to be stable, they are highly volatility in reality, which can be an issue that needs attention for refactoring or redesign. To test the fourth hypothesis, we show how to leverage our prediction approach to identify such issues.

As explained in Section 3, a design rule that dominates the change source can be recommended in the purely ACN-based approach, but incurs a penalty, pushing it down in the recommendation list. In the hybrid approach, this penalty

is negated if the design rule frequently changes with the change source, as reflected in the version history. In a traditional data mining approach, recommendations are solely based on the appearance frequency, and may always suggest that the design rules be changed with the change source, possibly also with other dependents of the design rules, instead of identifying this as an issue at all.

In the purely ACN-based approach, we can further highlight the violation of design rules that may indicate the deviation from designer’s intention by simply not recommending classes that are design rules of the change source. The deviation will be manifested by a low recall value of the prediction. By investigating the modification tasks for which these predictions generate low recall, we identified several problematic design issues, and our suspicions are confirmed by later versions of Hadoop when the design was indeed refactored and the problems we identified were addressed. We explain two examples in detail.

(1) Modification task #51, in version 0.1.0, describes changing the `DistributedFileSystem` class but not only is its parent class `FileSystem` impacted, another child of the `FileSystem` (`LocalFileSystem`) is also impacted. 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 UML dependencies between `DistributedFileSystem` and `LocalFileSystem`. When ignoring design rules, our model-based prediction does not recommend either `FileSystem` or `LocalFileSystem` and produced recall values low enough to draw our attention.

Upon further investigating of the source code for the three classes, we found an unusual design; the `FileSystem` class contains methods to construct both the `LocalFileSystem` and `DistributedFileSystem` classes. From a design rule theory perspective, the parent class `FileSystem` is a very unstable design rule and poor design because changes to a 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 0.19.0, the method to construct `DistributedFileSystem` had been deprecated in `FileSystem`, in favor of a method in an external class. It appears the design was refactored, due to the tight coupling of the unstable design rule.

(2) Modification task #1127 in version 0.12.0 is titled “Speculative execution and output of Reduce tasks” and it describes a change to the `ReduceTask` class (and only this class). When we examine the solution for this modification task, it also includes changes to 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 0.14.0, 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.

To verify this suspicion, we look forward in the version history and see, in fact, a refactoring occurs in version 0.19.0. Modification task #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.

Scenarios such as those identified by modification tasks #51 and #1127 show that our approach can be used to identify potential design issues that may require redesign or refactoring. By examining the modification tasks where the design rule based prediction produces low recall, we can potentially identify unstable design rules that are changing often. From this, the designer can decide whether to refactor the system in order to stabilize the design rule.

5. Discussion

Not all dependencies in a software system can be inferred from UML models. For example, in a model-view-controller (MVC) architectural pattern, the model and the view are semantically related, but are syntactically decoupled by the controller. When a new feature is added, both the view and the model often need to change together, but not the controller. If such dependencies dominate the software, our prediction quality may degrade.

Our current approaches do not take heterogeneous software artifacts into consideration, such as external configuration files. Our experiments on Hadoop show that the majority of change impacts can be captured by UML-based logical dependencies among classes. The results may be different for multi-language, multi-platform, distributed systems where not all dependencies can be picked up by UML class diagrams. For these systems, existing data mining

approaches [16, 17] may be more effective, given the existence of version history. On the other hand, the intermediate model we use, the augmented constraint network, is a general logic model independent of programming languages or platforms. Our ongoing work is to explore automated approaches of converting heterogeneous software artifacts into ACNs, so that the predictions can be applied to a broader range of software projects. Our future work is to evaluate our approaches on more systems, of varying sizes and types.

In our evaluation, we identified the change sources by reading the modification task descriptions in JIRA, which is limited in terms of accuracy and efficiency. Experienced developers would identify more accurate change sources and hence, yield different sets of recommendations. Performing a case study on a live system with developers identifying change sources and assessing the approaches’ recommendations remains a future work. As another option, Hill et al. [11] presented a work to automatically identify potential code structures from natural language queries. We are considering using Hill et al.’s work to identify change source files to see if it improves the efficiency.

The average precision shown in the case study is rather low, indicating that the developer may need to examine many irrelevant files. In this case study, the low precision stems from the fact that although each approach always recommends then files, the cardinality of the majority of solutions is less than or equal to five, as shown in Table 5. It would be better for our approach to determine the number of files to recommend based on the ACN and/or the version history. Improving our approach to incorporate variable number of recommendations is a future work.

6. Related Work

Software change impact analysis has been extensively studied from different perspectives [4, 5], mainly based on program analysis, historical data, or UML modeling. Approaches that use program analysis often use static slicing (e.g. Gallagher et al. [9]) or dynamic slicing (e.g. Agrawal et al. [1]). Dynamic slicing identifies impact based on the control-flow during an execution of the system for given inputs. Static slicing analyzes all possible control-flows in source code to identify impact. Our approaches differ from these existing approaches in that our approaches are based on the structure of logical *assumption* relations, which is a superset of direct UML relation and a subset of the transitive closure of UML relations. Our hybrid approach additionally considers information from version history.

Both Briand et al. [6] and Kung et al. [14] presented approaches to predict change impact on UML models, producing recommendations based on a set of detailed changes to the UML model (e.g. adding a specific method call from one class to another). Our approaches do not assume the

developer knows that level of detail regarding the change. Briand et al. assess impacts not only on class diagrams but also on other types of UML models. We only use class diagram relations in our approaches because detailed UML models are often not available in large systems, but the class diagram can be easily reverse engineered from the code base. Their work prioritizes the impacted classes based on distances between classes, our approaches additionally consider the stability of design rules and how many sub-tasks a file may participate in. Another major difference is that our approach can leverage data mining techniques to better facilitate maintenance tasks.

Our work is related to data-mining-based change impact analysis because our hybrid approach makes use of such a technique. Zimmermann et al. [17] and Ying et al. [16] are representative works in data-mining impact analysis. Both approaches mine the version history for sets of files that change together (change patterns) and recommend files in the change patterns that occur more than a constant number of times (minimum support). A major difference between our work and theirs is that our approaches do not rely solely on version history, and can be applied to relatively young systems. While their work also identifies “surprising” couplings that usually happen among heterogenous files, our approaches identify design issues from the deviations between purely ACN-based prediction and real change sets.

7. Conclusion

In this paper, we presented two approaches to predict the impact scope of a given change to facilitate software maintenance activities. The prediction approaches were based on the logical relation among classes extracted from UML class diagrams. The first approach exclusively used the logical model to make recommendations while the second, hybrid approach additionally took change set information into consideration. We evaluated our approaches by applying them on the first 14 minor releases of the open source Hadoop project, and compared the quality of the prediction with that of an existing data mining technique. The results showed that both our approaches produce better predictions when the system is immature and the version history is not well-established, and our hybrid approach produces comparable results with data mining as the system evolves. In addition, we showed that by investigating deviations between the purely model-based predictions and the actual change set, we can potentially identify issues that suggest redesign or refactoring.

References

- [1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.
- [2] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *9th 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 *18th 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, Y. Labiche, and L. O’Sullivan. Impact analysis and change management of UML models. In *19th International Conference on Software Maintenance*, pages 256–265, Sept. 2003.
- [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. Modularity analysis of logical design models. In *21st IEEE/ACM International Conference on Automated Software Engineering*, pages 91–102, Sept. 2006.
- [9] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.
- [10] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Nov. 1994.
- [11] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with Dora to expedite software maintenance. In *22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 14–23, Nov. 2007.
- [12] 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>.
- [13] S. Huynh, Y. Cai, Y. Song, and K. Sullivan. Automatic modularity conformance checking. In *30th International Conference on Software Engineering*, pages 411–420, May 2008.
- [14] D. C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *10th International Conference on Software Maintenance*, pages 202–211, Sept. 1994.
- [15] Lattix Inc. The Lattix approach whitepaper. <http://www.lattix.com>, 2004.
- [16] 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.
- [17] T. Zimmermann, P. W. gerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering*, pages 563–572, May 2004.