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.

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. The limitation of only using syntactic dependencies to predict change impact has been recognized. A recent direction is to leverage historical data [9, 10] to make recommendations based on how frequently two parts of the system have changed together. The problem is that the accuracy of these approaches relies on the existence of well-established version histories. Their effectiveness therefore degrades when the project is relatively new or the design is refactored [9].

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. The approaches are based on the *assumption* relation among design decisions of a software project, represented by a logical framework called the *augmented constraint network* (ACN) [5, 6].

An ACN can express relations among classes, modules, aspects, as well as environmental conditions (e.g. memory size, requirements [5, 6]). As the first paper to explore the ability of ACN modeling to predict change impact, we only consider the assumption relation among classes in object-oriented software—the assumptions that can be automatically derived from UML class diagrams.

We observe that the assumption relation derived from a UML class diagram reveals a lot more dependencies than the results from syntactic analysis but fewer dependencies than the transitive closure. Leveraging the assumption relation, we contribute an algorithm to predict change impact. It is possible that some dependencies can be manifested easily using data mining or other more complex techniques, but not captured by either syntactical or assumption relations. We thus complement the purely ACN-based prediction approach with the knowledge obtained from version history to improve prediction accuracy.

We evaluate our approaches (Section 3) on 14 versions of the open source Hadoop[1] 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.

[1] http://hadoop.apache.org/core/

2. Prediction Approaches

Our approaches start from extracting and formalizing logical assumption relations implied by UML class diagrams into a logical framework called the *augmented constraint network* (ACN) [5, 6]. An ACN consists of a *constraint network* that models design decisions and their relations, and a *dominance relation* that formalizes the design rules [1, 5]. 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* relations among design variables.

As the input to our predicting approaches, a UML model can be either constructed by the designer or reverse engineered from source code. Our approach currently supports translating of classes, interfaces, and the major binary relations of class diagrams such as generalization and realization into ACN [7]. For example, from a decision-making perspective, each class consists of two design dimensions: an interface dimension (not to be confused with the interface construct found in many object-oriented languages), 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.

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 augment the constraint network with a binary *dominance relation* to model asymmetric dependence relations among decisions, the essence of design rules. For example, the last line shows that the decision of `A`'s implementation cannot influence the design of its interface. In other words, we cannot arbitrarily change `A`'s public interface to simplify the class's implementation because other components may rely on that interface. 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)
```

To assess change impact, we consider the system's subsystems (features, functions), and assess how many subsystems a variable is involved in. The approach is based on Cai et al.'s [6] 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. 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 and generates a set of sub-ACNs each containing all the decisions needed to accomplish a subsystem.

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.

The prediction algorithm works on the condensation graph generated from the ACN. The first step is to figure out all the the classes that are logically related to the change source from the graph. Instead of recommending all these classes as equally likely to be impacted the source, we rank the impact likelihood of each class by assigning weights to them. We first assign a weight to a variable according to the number of sub-ACNs (as identified by Cai et al.'s [5, 6] algorithm) involving it. Since a sub-ACN represents all the decisions needed to accomplish a particular 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.

We further modify the weight assignment based on the following major factors: first, the closer a variable to the source, the more likely it will be impacted by the source. Second, although edges from subordinating decisions to design rules are removed from an graph, subordinating decisions do occasionally affect design rules in reality. We lower the weights of the design rules such that the further away the design rule is from the source, the less likely it will be influenced and changed. Therefore, allowing design rules to be recommended, but with a penalty to indicate that changing design rules is not as common as changing non-design rules. 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.

Hybrid Prediction. It is possible that there are some dependencies that cannot be picked up by assumption relations, and that some variables change together more often than others even though they have similar probability to change based on their assumption relations. We thus propose a hybrid algorithm that feeds the purely ACN-based approach with knowledge obtained from version history. The idea is to find how often a variable changes concurrently with the change source. Zimmermann et al. [10] refer to this ratio as the *confidence* level. After that, we adjust the weights resulting from the purely ACN-based algorithm by multiplying the weight of each variable by the ratio from data mining.

3. Preliminary Evaluation

We choose Hadoop, an open source map/reduce system for distributed computing written in Java, to evaluate our approaches because it is relatively young with only three years of development history and no major releases. 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.

We use the standard metrics of *precision*, *recall*, and F_1 to measure the quality of predictions. Mathematically, given a change source f_S and solution f_{sol} for a modification task m, we define:

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

For each modification task, we manually examine the task description to identify one to three change sources. Most of the modification tasks were initiated from only one class. We first reverse engineered the code base of all 14 releases into 14 UML class diagrams (since prescriptive class diagrams were unavailable), and then translated the UML models into 14 ACNs. 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 [10]. Also following their work, we only report the top ten files recommended by each approach. 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.

Evaluation Results Table 1 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 Hybird 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.

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 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. 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 ACN's 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.

4. Related Work

Software change impact analysis has been extensively studied from different perspectives [2, 3]. Both Briand et al. [4] and Kung et al. [8] 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. Briand et al. assess impacts not only on class diagrams but also on other types of UML models. Their work also prioritizes the impacted classes based on distances between classes. Our approaches additionally consider the stability of design rules and how many subsystems a file may participate in. Our approaches are based

Table 1. Evaluation Results

Version	Precision			Recall			F_1		
	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

on the structure of logical *assumption* relations, which is a superset of direct UML relations and a subset of the transitive closure of UML relations.

Our work is related to data-mining-based change impact analysis because our hybrid approach makes use of such a technique. Zimmermann et al. [10] and Ying et al. [9] 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). Different from their work, our approaches do not rely solely on version history, and can be applied to relatively young systems.

5. Conclusion and Future Work

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.

Our ongoing work is to explore automated approaches of converting heterogenous software artifacts into ACNs, so that the predictions can be applied to a broader range of software projects, taking heterogenous software artifacts into consideration. In this case study, the low precision stems from the fact that although each approach always recommend ten files, the cardinality of the majority of solutions is less than or equal to five. Improving our approach to incorporate variable number of recommendations based on ACN and/or the version history is a future work.

References

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

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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.