



University of Zurich
Zurich Open Repository and Archive

Winterthurerstr. 190
CH-8057 Zurich
<http://www.zora.uzh.ch>

Year: 2008

A Bayesian Network Based Approach for Change Coupling Prediction

Zhou, Y; Wuersch, M; Giger, E; Gall, H C; Lue, J

Zhou, Y; Wuersch, M; Giger, E; Gall, H C; Lue, J (2008). A Bayesian Network Based Approach for Change Coupling Prediction. In: Working Conference on Reverse Engineering, Antwerp, Belgium, 15 October 2008 - 18 October 2008, 27-36.

Postprint available at:
<http://www.zora.uzh.ch>

Posted at the Zurich Open Repository and Archive, University of Zurich.
<http://www.zora.uzh.ch>

Originally published at:
Working Conference on Reverse Engineering, Antwerp, Belgium, 15 October 2008 - 18 October 2008, 27-36.

A Bayesian Network Based Approach for Change Coupling Prediction

Abstract

Source code coupling and change history are two important data sources for change coupling analysis. The popularity of public open source projects in recent years makes both sources available. Based on our previous research, in this paper, we inspect different dimensions of software changes including change significance or source code dependency levels, extract a set of features from the two sources and propose a bayesian network-based approach for change coupling prediction. By combining the features from the co-changed entities and their dependency relation, the approach can model the underlying uncertainty. The empirical case study on two medium-sized open source projects demonstrates the feasibility and effectiveness of our approach compared to previous work.

A Bayesian Network Based Approach for Change Coupling Prediction

Yu Zhou^{1,2}, Michael Würsch², Emanuel Giger², Harald Gall², Jian Lü¹

¹State Key Lab. for Novel Software Technology, Nanjing University, China

²Software Evolution and Architecture Lab., University of Zurich, Switzerland
{zhou,wuersch,giger,gall}@ifi.uzh.ch, lj@nju.edu.cn

Abstract

Source code coupling and change history are two important data sources for change coupling analysis. The popularity of public open source projects in recent years makes both sources available. Based on our previous research, in this paper, we inspect different dimensions of software changes including change significance or source code dependency levels, extract a set of features from the two sources and propose a bayesian network-based approach for change coupling prediction. By combining the features from the co-changed entities and their dependency relation, the approach can model the underlying uncertainty. The empirical case study on two medium-sized open source projects demonstrates the feasibility and effectiveness of our approach compared to previous work.

1. Introduction

Continuing change is one of the accompanying phenomena along with software evolution [17]. The fact that huge costs accumulate during the evolution phase stresses the importance of research on the underlying changes [1]. The public availability of open source project repositories provides a large data set for various research, including change-coupling analysis. Mining these repositories to guide future changes has become a common practice [19] in software evolution research. Similar to the ideas presented by Zimmermann *et al.* in [30], Developers conducting software maintenance tasks can be augmented by tools that for example provide a warning that *if you apply a change to this entity, you will probably also have to propagate it to that entity later*. This information is also helpful during program comprehension in general and reverse engineering in particular, as a programmer can quickly get an impression of the dependencies between the parts of a system in terms of change couplings.

Existing work on change-coupling typically leverages change history information from version control systems to

extract co-change patterns. These are sets of files that frequently changed together in the past, *e.g.*, they often appear in the same CVS commit action [9]. However, the types of changes, which usually have different implications on the co-change probability of other related entities, are not considered. For example, the indentation formatting and class restructuring definitely have a different change impact on other entities. Static dependencies within the code are another source of heuristics to guide change coupling analysis. For instance, adding new methods to abstract classes or interfaces will very likely lead to co-changes along the inheritance hierarchy or in the implementing classes.

Change history offers a valuable source for future change prediction. Existing work in this area tends to omit two important dimensions concerning change-couplings, *i.e.*, the *author* that was responsible for a particular change and the *exact time* when the change happened. Different developers are usually assigned with different roles for particular kinds of software entities. This is especially the case for medium or large-scale software development with numerous developers contributing to one system. Due to such code-ownership mechanisms, changes applied by a particular programmer are more likely to propagate between the entities of his responsibility than to those of others. Changing a software system is a continual activity. More recent changes to a part of the system reflect that it is/was under active development and therefore likely to change again in the near future [12].

Neglecting the information mentioned above inevitably reduces the prediction performance. In recent case studies, this fact often reflects in the imbalance between precision and recall rates, as seen for example in [20, 29, 30]. In order to achieve a high value of one of either rates, the other usually has to be sacrificed.

The cause-effect analysis of change coupling coincides with the idea of a *Bayesian network* (a.k.a., *probabilistic cause-effect model*) which is based on conditional probabilistic theory. Due to its ability to reveal the underlying causality, the Bayesian network has a wide range of applications for modeling the uncertainty [5, 6, 20, 27].

The above observations motivate us to reconsider the process of analyzing software changes by investigating what kind of information provided by current version control systems are most relevant to change couplings. We incorporate these into a Bayesian network based approach to model the uncertainty, which allows us to infer possible co-changes between entities in the future.

More formally expressed, given a set of entities $S_e = \{e_1, e_2, \dots, e_n\}$ that belong to a software system and given that a change was applied to some entity $e_i \in S_e$, this paper presents an approach to predict the set of other entities $S_c = \{S_c \subset S_e \mid e_i \notin S_c\}$ that are also likely to change. Our work makes the following contributions:

1. We present a change propagation model based on a Bayesian Network that incorporates static source code dependencies as well as different features extracted from the release history of a system, such as change type significance and author information.
2. In a case study with two open source systems, we demonstrate that our change propagation model is able to predict future change couplings. By incorporating evolutionary data, we are able to outperform similar existing approaches significantly.

The remainder of this paper is organized as follows: In Section 2 some background information on change type significance levels, and the Bayesian network model is given. Section 3 explains our approach in detail. We evaluate the proposed approach in Section 4 by means of two open source case studies, *i.e.*, Azureus¹ and ArgoUML². A discussion of the results and also of the threats that might affect the validity our approach is held in Section 5. We present related work in Section 6. Section 7 concludes our work.

2. Background

In this section, we give a short introduction to change type significance and Bayesian network.

2.1. Change Type Significance

Common version control systems, including CVS, store changes between subsequent versions usually on a diff-based textual level and therefore even simple indentations can generate a change record. Aware of this, *distilling change significance according to different types is needed to improve co-change analysis* [25].

Our *taxonomy of source code changes* allows a more differentiated view on source code changes, as it is based on

tree edit operations on the abstract syntax tree rather than on textual line changes. The atomic operations include *insert*, *delete*, and *substitute* based on which *move*- and *update*-operations can be derived. The significance level (*e.g.*, low, medium, high, crucial) denotes the level of impact of a change operation on other source code entities. To assess the significance level, changes are classified into two gross categories: body-part changes and declaration-part changes. The former includes method body changes, structure statement changes, and class body changes; while the latter includes access modifier changes, final modifier changes, attribute declaration changes, method declaration changes, and class declaration changes. The basic metrics for mapping these change types to different significance levels are based on complexity indicators (*e.g.*, nesting depth), or whether they are functionality-modifying or functionality-preserving. For details, we refer to our previous work [8, 9, 10].

2.2. Bayesian Network

A Bayesian network is a directed acyclic graph model that represents conditional independencies between a set of variables [22]. It has two constituents: One is a network graphical structure which is a directed acyclic graph with the nodes of variables and arcs of relations. The other is the conditional probability table associated with each node in the model graph. Machine learning techniques are able to estimate the structure and the conditional probability table from the training data. The table represents the conditional probability distribution of the associated graph nodes. Based on the Bayesian probability inference, the conditional probability can be estimated from the statistical data and propagated along the links of the network structure to the target label. By setting a threshold of confidence, the final probability value can be used as the indication for the classification decision.

The Bayesian formula can be mathematically expressed as below:

$$P(H_j | \vec{E}) = \frac{P(\vec{E} | H_j) \times P(H_j)}{\sum_{i=1}^n P(\vec{E} | H_i) \times P(H_i)} \quad (j = 1, 2, \dots, n) \quad (1)$$

According to the basic statistical theory, *e.g.*, the Chain Rule and independency relation derived from the network structure, the joint probability of \vec{E} can be calculated by the production of local distributions with its parent nodes, *i.e.*,

$$P(\vec{E}) = \prod_{i=1}^n P(E_i | \text{ParentOf}(E_i)) \quad (2)$$

In the above formulas, \vec{E} denotes a set of variable values, *i.e.* $\vec{E} = \{E_1, E_2, \dots, E_n\}$. H is termed as hypoth-

¹<http://azureus.sourceforge.net/>

²<http://argouml.tigris.org/>

esis. $P(H)$ is called the prior probability and $P(H|\vec{E})$ is called posteriori probability of H given \vec{E} . If E_i has no parent nodes, $P(E_i|ParentOf(E_i))$ is equal to $P(E_i)$.

In our case, we use the Bayesian network model to answer the following question: *given the features that belong to two software entities, for example names, authors, change significance levels, etc., are they likely to co-change in a revision transaction or not?*

3. Approach

Our approach consists of three sequential steps:

1. *Data Import & Preprocessing* includes extracting the change history information and the reconstruction of the transactions from the version control repository [23], transforming the selected source releases into FAMIX model [4], and extracting the fine grained source change information [10]. This data is used to build the Release History Database (RHDB) [7].
2. *Feature Extraction and Instance Generation* involves selecting features that influence on whether two entities might co-change. These features and their concrete values—features instances—are used to build the network in the next step.
3. *Training & Validation* uses the selected entities and their instances to feed and build up the network structure, then calculates the conditional probability values to predict future co-changes. Features are the nodes of the Bayesian network and feature instances are entries in the probability table of their associated node.

We explain each step in detail in this section. Figure 1 illustrates our approach.

3.1. Data Preprocessing

While version control systems clearly provide an invaluable aid in the team development process, they often show severe deficiencies when it comes to accessing the historical data for software evolution analysis. These deficiencies make additional data extraction and processing steps necessary which can be automated by using EVOLIZER, our software evolution research and analysis platform. EVOLIZER basically stems from the idea of having a Release History Database (RHDB) [7] that integrates information originating from various repositories, such as CVS and Bugzilla, in a single database. In the following we describe the three tasks that are necessary 1) to infer the transactions that give information about the co-change relationships of the files under version control, 2) to calculate the static source code dependencies, and 3) to extract and classify the source code

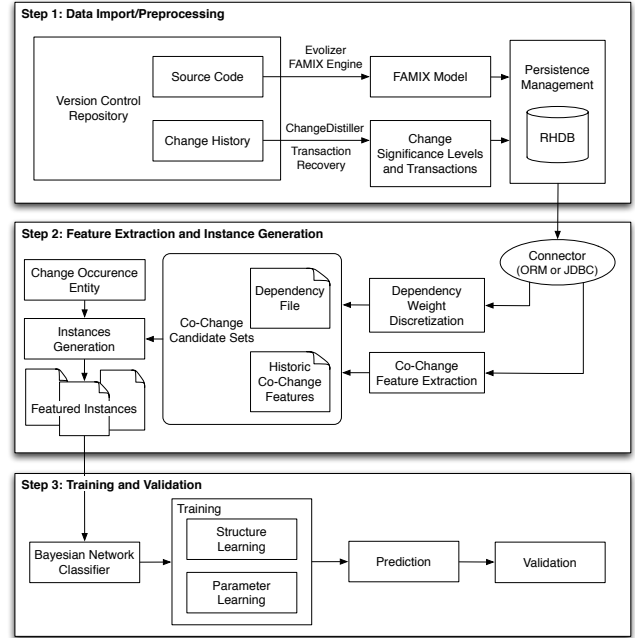


Figure 1. The Change Prediction Process

changes into change types which allows to assess their significance level in terms of the impact of the changes on other source code entities.

Task 1: Co-changes. CVS provides only file-based versioning but does not store information about change sets or transactions, *i.e.*, about the set of files that were changed together. To recover the transactions, EVOLIZER retrieves all modification reports from CVS and uses a sliding window algorithm to group them [7, 30]. The result is then stored in the RHDB for further processing.

Task 2: Static Source Code Dependencies. While the previous task is able to reveal implicit and evolutionary dependencies, so called *logical couplings*, the second task complements this data with information derived from an analysis of the static dependencies within the source code, *e.g.*, caller-callee or inheritance relationships. For that, EVOLIZER builds a FAMIX model [4] for all releases (or any snapshot, if configured so) of a system, based on the source code retrieved from CVS, and stores it in the RHDB. FAMIX is a language-independent source code metamodel that has proved itself useful in many applications, *e.g.*, in [16, 24].

Task 3: Change Significance Levels. CHANGEDISTILLER extracts source code changes based on the algorithm presented in [10] and classifies them according to a *taxonomy of source code changes* into *change types* [8]. For that, it uses EVOLIZER to interface the RHDB and applies tree differencing pairwise on abstract syntax trees generated from subsequent revisions of the source code files that con-

tribute to the system under investigation. The fine-grained change data, including the change classification, are also stored in the RHDB for later use.

After these three tasks are completed, we have all the data about the change history of the system under investigation at our disposal. Next we explain in detail which features we extract from the data generated so far and how we feed them into a Bayesian network in order to be able to predict future changes.

3.2. Feature Extraction and Instance Generation

This step is concerned with identifying features that are related to co-change events. We extract the concrete instances of these features from the RHDB that we have built during the data preprocessing phase described in the last section.

Instead of simply giving considerations to past co-change occurrence, we choose a set of factors that most likely have an effect on the co-change relation between entities, *e.g.*, these are features that cause a certain set of files being often changed together and then committed in the same transaction to a CVS repository. The features are *static source code dependency level*, *co-change frequency*, *change significance level*, *age of change*, and *author*. We consider these to be good indicators for co-change occurrence because:

Source Code Dependency Level. Static source code dependencies, such as message passing between classes, inheritance, or interface implementations can influence the co-change behavior of source code entities. For instance, adding a new method to an interface will propagate to all implementing classes. We consider two kinds of dependencies: associations between classes (denoted D_DEP_LEVEL), in particular method calls, and hierarchical dependencies (denoted H_DEP_LEVEL), such as inheritance and interface implementation. We count the number of each kind of dependencies between two source code entities and discretize them into different levels: *low*, *middle*, *high* and *uncertain*. Discretization is done as follows: *uncertain* expresses the absence of any direct dependency. In case of the D_DEP_LEVEL, the total number of dependencies of the given entity is divided into the three intervals *low*, *middle*, *high*. For example, given three packages p_1 , p_2 , and p_3 , if there are 90 calls from p_2 to methods in p_1 and 10 calls from p_3 to p_1 , the total number of dependencies on p_1 is 100 and we classify the dependency between p_1 and p_2 as *high*, and the dependency between p_1 and p_3 as *low* respectively. In case of H_DEP_LEVEL, values above average are classified as *high*, all others as *low*.

Co-Change Frequency. This reflects how many times entities changed together in the past. It is a commonly used factor in literature on change prediction, for example in [20,

29, 30]. We discretize the co-change frequency for each pair of entities e_1 and e_2 as follows: We calculate the number of transactions that contain both, e_1 and e_2 , and divide this value by the total number of transactions that contain e_1 . If the result is above the threshold 0.5, we classify the co-change frequency as *high*, otherwise as *low*.

Change Significance Level. In [8], Fluri and Gall assigned a significance level to source code changes in terms of the change impact on other source code entities. For instance, changing the signature of a method will most likely affect all of its callers whereas renaming a local variable only affects its local scope. According to their classification, we calculate for each change whether its significance level is *low*, *middle*, *high*, or *crucial*.

Age of Change. Age of change considers the point in time of a change. If the changes are old, it implies that an entity is more stable. Therefore recent changes indicate a more change prone entity in the near future. We divide the whole project life span into four periods of equal length and classify changes depending on the period that they took place in into *old*, *middle*, *new*, or *latest*.

Author. Taking the author of a change into account is based on organizational aspects of large-scale distributed software development with multiple participants. Usually a developer works on her assigned entities as a work package or in terms of code ownership.

The purpose of our approach is to predict co-changes. Using the Bayesian network model and given a change in a specific entity, we are able to say how likely it is that a certain other entity has to be changed in conjunction. We need three other features to incorporate this goal in the network.

Change request. This denotes the entity that initially changes due to a request, *e.g.*, because there is a need to fix a bug, apply refactoring measures, etc.

Change candidate. The change candidate is the entity for which we predict the probability that it will change together with the entity of the *change request*.

Co-Change. This is the target label we want to predict. It has the instances *yes* or *no* to indicate whether or not the candidate entity changes together with the change request or not. The final feature set can be found in Table 1.

We take the following procedure to generate the instances (the algorithm is also outlined in Algorithm 1):

1. Trace the revisions in the RHDB, get the change entity in a transaction with their change age, author name and change significance level. This information can be retrieved from the RHDB database (*Lines: 7 to 8*).
2. Select candidates from the their dependent set, and their dependency level (*Lines: 9 to 10*).
3. Check their past co-change frequency (*Lines: 11 to 14*).
4. Check whether they co-changed in this transaction. If it is true, label it with 'yes', otherwise 'no', and update the co-change historical frequency (*Lines: 15 to 23*). The change frequency can be

```

Data: rreq:revision request
Result: InstanceList
1 begin
2    $InstanceList \leftarrow \emptyset$ ;
3    $depList \leftarrow dependentSetOf(rreq)$ ;
4    $pastList \leftarrow pastCochangeSetOf(rreq)$ ;
5    $transList \leftarrow transactionSetOf(rreq)$ ;
6    $ins$ : Instance;
7   Assign  $rreq$  to  $ins.changerequest$ ;
8   Retrieve and assign  $rreq$ 's author name, change age and
   significance level features to  $ins$ ;
9   for  $r_i \in depList$  do
10    Assign  $dependLevel$  to  $ins.dependlevel$ ;
11    if  $r_i \in pastList$  then
12       $ins.pastchangelevel \leftarrow pastFrequency(r_i)$ ;
13    else
14       $ins.pastchangelevel \leftarrow no$ ;
15    if  $r_i \in transList$  then
16       $ins.label \leftarrow yes$ ;
17      if  $r_i \notin pastList$  then
18        add  $r_i$  to  $rreq$ 's  $pastList$ ;
19         $updateCoChangeRateSet(rreq, r_i)$ ;
20    else
21       $ins.label \leftarrow no$ ;
22      if  $r_i \in pastList$  then
23         $updateCoChangeRateSet(rreq, r_i)$ ;
24     $InstanceList \leftarrow InstanceList \cup \{ins\}$ 
25  for  $r_i \in pastList - depList$  do
26     $ins.dependlevel \leftarrow uncertain$ ;
27     $ins.pastchangelevel \leftarrow pastFrequency(r_i)$ ;
28    if  $r_i \in transList$  then
29       $ins.label \leftarrow yes$ ;
30       $updateCoChangeRateSet(rreq, r_i)$ ;
31    else
32       $ins.label \leftarrow no$ ;
33       $updateCoChangeRateSet(rreq, r_i)$ ;
34     $InstanceList \leftarrow InstanceList \cup \{ins\}$ 
35  for  $r_i \in transList - (pastList \cup depList)$  do
36     $ins.dependlevel \leftarrow uncertain$ ;
37     $ins.pastchangelevel \leftarrow no$ ;
38     $ins.label \leftarrow yes$ ;
39    add  $r_i$  to  $rreq$ 's  $pastList$ ;
40     $updateCoChangeRateSet(rreq, r_i)$ ;
41     $InstanceList \leftarrow InstanceList \cup \{ins\}$ 
42 end

```

Algorithm 1: Feature Extraction and Instance Generation

Feature Name	Value(s)
CHANGE_REQUEST	entity name
AGE_OF_CHANGE	{old, middle, new, latest}
AUTHOR	author name
SIG_LEVEL	{low, middle, high, crucial}
D_DEP_LEVEL	{uncertain, low, middle, high}
H_DEP_LEVEL	{uncertain, low, high}
CHANGE_CANDIDATE	candidate name
PAST_CO.CHANGE.LEVEL	{none, low, high}
CO.CHANGE.LABEL	{yes, no}

Table 1. List of Features

calculated by the co-change times divided by the transaction numbers. 5. Select candidates from co-change history but not included in the dependency. Since it has no dependency level, assign the dependency level with ‘uncertain’ (Lines: 25 to 26). 6. Check their past co-change frequency, assign the co-change rate to the instance, and label them for this transaction (Lines: 27 to 33). 7. Select those entities that co-changed in this transaction but not included in the above two sets, label it ‘yes’, add it to the historical co-change list and update the accompanied frequency rate (Lines: 35 to 40).

3.3. Bayesian Network Training and Prediction

Now that we have generated the set of instances, they are used to build the Bayesian network. For evaluation purpose, the set is divided into two parts: one is considered to be the training set, the other is used for prediction. By *training*, we mean using the entities in the training set to learn the structure and the parameters of the Bayesian network. By *prediction*, we mean using the trained Bayesian network to predict the remaining part of the dataset and evaluate the precision rate as well as recall rate. As the instances are populated from the repository, each candidate is labeled as *yes* or *no* for the co-change occurrence in the same transaction with the initial change request entity. Thus this is a supervised learning process.

The practical Bayesian network application emerged much later in contrast to the relatively long history of Bayesian network theory. This is primarily due to the expensive computation requirement [2].

We use the K2 algorithm developed by Cooper *et al.* to overcome this limitation. K2 is a greedy search based learning algorithm. It begins with the assumption that a node has no parents, and then adds incrementally the parent whose addition increases the probability of the resulting structure the most. By using this heuristic, the complexity is reduced to polynomial time [3]. It is widely used as a classical structure learning technique in Bayesian networks, and known for its good performance [18].

When structure learning is completed, the network parameter learning process (*i.e.*, the conditional probability table associated with the node) starts. We use the SimpleEstimator algorithm implemented in WEKA [28] which calculates the feature frequency values from training sets and deduces the corresponding probability according to Formula 1 and 2 in Section 2.2.

4. Case Study

The selection criterion for our case study is based on the following considerations:

1. Medium to large scale open source projects that provide public access to source code and use a versioning system that allows us to extract the historical information;
2. Projects with multiple contributors and a relatively long history, *i.e.*, at least three years;
3. Projects with large group of users and broad application domain in practice;

Azureus and ArgoUML fit these requirements well. The former is one of the most popular BitTorrent clients with the first release in June 2003. The latter is a well known UML modeling application with the first release in July 2002. Both are written in Java, have a public accessible CVS repository, and are still under development.

Mirarab *et al.* proposed a Bayesian network based approach to predict change couplings in [20]. They built three kinds of Bayesian network models, each having a different focus and used Azureus as a case study to evaluate their approach. Their network models omit several features that, as we argued in Section 3.2, have implications on the co-change behavior. Our results show including these features yields a better performance; we use Azureus to conduct a comparison experiment between their approach and ours. Mirarab *et al.* used Java packages as granularity level for their case study, so we choose the same to guarantee consistency and comparability between the two approaches.

4.1. Experimental setup

For the case of Azureus, we use the same releases that Mirarab *et al.* explicitly listed in their paper (see Table 1. *Measures of Azureus2* in [20]), except Release 2.0.0.8 which was not available anymore, to extract the package dependency information. Mirarab *et al.* only provided average performance values over the whole history, therefore we also calculated average values to establish the comparison as shown in Figure 2. To demonstrate that our approach also shows good prediction performance for other systems, we selected five releases from ArgoUML. Both, the selected releases of Azureus, as well as those of ArgoUML, are listed in Table 2.

Azureus2		ArgoUML	
Version	Date	Version	Date
2.0.3.2	2003.10.12	0.10.1	2002.10.09
2.0.7.0	2004.01.31	0.12	2003.08.18
2.1.0.2	2004.06.20	0.16	2004.07.19
2.3.0.2	2005.05.25	0.181	2005.04.30
2.5.0.0	2006.08.21	0.22	2006.08.08

Table 2. Information for Azureus 2 and ArgoUML

For each of the two projects, we populate a RHDB and perform the data preprocessing tasks that we outlined in Section 3.1, that is, we recover the transactions to get the co-change information, build a FAMIX model for each release to get the static source code dependencies, and calculate the change significance levels to take the nature of the change into account. Then we extract the features from the RHDB and generate the instances (Section 3.2) to train our Bayesian network (Section 3.3) and evaluate its prediction performance. This evaluation is explained in the following sections.

4.2. Metrics

In [20], Mirarab *et al.* used two sets of evaluation criteria, *i.e.*, point-biserial correlation and information retrieval metrics. As we need a way of measuring the accuracy of our prediction result, we are interested in the *precision* and *recall* rates. Using these two metrics as evaluation criteria is a common procedure in related work, *e.g.*, in [13, 26, 29, 30] and the areas of both information retrieval and statistics.

Precision is used to measure the exactness of the prediction set, while recall evaluates the completeness. Precision and recall can be expressed mathematically:

$$precision = \frac{TruePositives}{TruePositives + FalsePositives} \quad (3)$$

$$recall = \frac{TruePositives}{TruePositives + FalseNegatives} \quad (4)$$

Based on the precision and recall, we can calculate the F-measure:

$$f\text{-measure} = 2 \times \frac{precision \times recall}{precision + recall} \quad (5)$$

In information retrieval the F-measure denotes the balance and discrepancy between precision and recall.

4.3. Evaluation

Like Mirarab *et al.* in [20] and Ying *et al.* in [29], we split the data set along the time line into a training and a validation set. Then we use the first set to train the Bayesian network and evaluate whether it predicts the validation set well.

For Azureus, as well as for ArgoUML, we run the Algorithm 1 for the change entities in the change history and generate instances for the subsequent releases. The static source code dependencies are calculated once, that is at the beginning of the interval. The instances are generated by iterating over the change history. Mirarab *et al.* did not provide the split percentage between the training and validation set in their work [20]. To get the average prediction result,

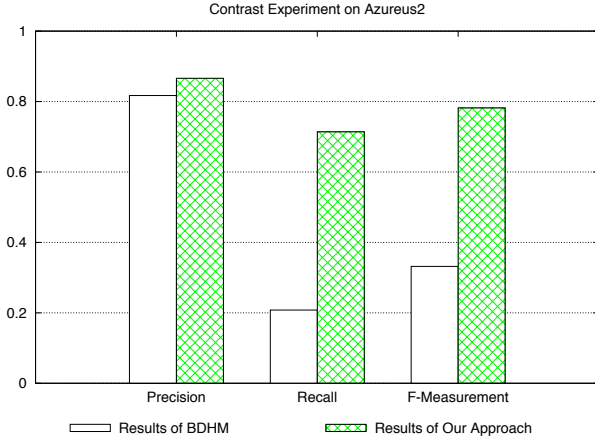


Figure 2. Comparison with previous work on Azureus2

we perform nine experiments for each interval by adjusting the training split from the first 10% to the first 90%, in steps of 10% each time. And in this way, we sum up the results and calculate the average precision and recall.³

In each of the nine experiments, after learning from the training set, we look at each co-change that actually took place in the validation set. If we predicted the co-change, we consider it as a *true positive*. If we did not predict it, we count it as *false negative*. In case we predict a co-change that actually did not happen, *i.e.*, that is not in the validation set, it is counted as *false positive*. The results of the experiments for the two projects can be found in Table 4. As we are more interested in the prediction performance for actual co-changes, we provide the prior probabilities for ‘yes’ in each interval.

The average weighted precision and recall rate can be calculated by:

$$avg_pre = \frac{1}{\sum_{i=1}^n N(i)} \times \sum_{j=1}^n \left(\left(\frac{1}{m} \times \sum_{k=1}^m pre_k \right) \times N(j) \right) \quad (6)$$

$$avg_rec = \frac{1}{\sum_{i=1}^n N(i)} \times \sum_{j=1}^n \left(\left(\frac{1}{m} \times \sum_{k=1}^m rec_k \right) \times N(j) \right) \quad (7)$$

In Formula 6 and 7, m denotes how often training was carried out, which is nine times in our case; n denotes number of intervals, which is four in Azureus2, as well as in ArgoUML. $N(j)$ denotes the number of instances in interval j . By using Formula 5, an average F-measure can be derived. The weighted average results are listed in Table 3.

	Precision	Recall	F-measurement
Azureus2	0.866	0.714	0.782
ArgoUML	0.757	0.532	0.625

Table 3. Average Experiment Result

The comparison result of our work with [20] is illustrated in Figure 2. In [20], the authors proposed three different Bayesian models among which the *Bayesian Dependency and History Model (BDHM)* comes closest to ours. Figure 2 depicts how our approach performs in contrast to the BDHM. The average performance result of the five releases of Azureus are more accurate, raising the recall rate from the level of 20% to the level of more than 70%, and the precision rate from 82% to 87% respectively.

5. Discussion

Software co-changing is an activity triggered by a mix of accidental and essential causes. By extracting the features from this activity, the Bayesian network can be used to model the underlying uncertainty and predict future change couplings. The experimental results for the two open source systems demonstrate the feasibility and effectiveness of our approach. In particular, our comparison with existing work shows that incorporating information about static source code dependencies, author information and about the nature of changes, *i.e.*, their impact on other source code entities, can leverage the change coupling prediction performance in terms of precision and recall significantly.

All the features needed for our algorithm are available from the version control repositories, so our approach can be used to extend or augment existing tools in order to provide for example recommendations of the kind “If you change this package, then you will probably also have to consider changing these packages...” to the developer. The evaluation results of our case study show that, even when considering the early stage of software development where only few training data is available, we achieve a precision rate of 60% and a recall rate of 40%. The overall average performance increases up to 80% for precision, and 60% for recall respectively, once the training set increases and the initial training phase is over.

An apparent limitation of existing approaches that focus on transaction-based change coupling solely, emerges from the fact that most of the current mainstream version control systems only allow us to identify the co-changed entities, *i.e.*, entities that appear in the same transaction—either directly by using available API (in case of SVN) or by applying some heuristics and performing additional computation steps (in case of CVS). They do not provide us information about the logical change ordering [30]. We can not induce solely from version history which change triggered logi-

³The data used for the case study can be downloaded from our website: ‘http://seal.ifi.uzh.ch/fileadmin/User_Filemount/reagle/data.tgz’

TP ^b	Azureus 2.0.3.2 to 2.0.7.0 (25278 instances, 16.2% ^a)			Azureus 2.0.7.0 to 2.1.0.2 (41246 instances, 16.8%)			Azureus 2.1.0.2 to 2.3.0.2 (127555 instances, 11.3%)			Azureus: 2.3.0.2 to 2.5.0.0 (343264 instances, 14.5%)		
	Precision	Recall	F-M ^c	Precision	Recall	F-M	Precision	Recall	F-M	Precision	Recall	F-M
10%	0.681	0.426	0.524	0.760	0.640	0.695	0.641	0.523	0.576	0.908	0.779	0.839
20%	0.773	0.471	0.585	0.800	0.537	0.642	0.767	0.531	0.628	0.905	0.785	0.841
30%	0.833	0.501	0.626	0.827	0.555	0.664	0.781	0.552	0.647	0.906	0.789	0.844
40%	0.841	0.526	0.647	0.854	0.571	0.685	0.792	0.570	0.663	0.904	0.791	0.844
50%	0.863	0.538	0.663	0.881	0.589	0.706	0.793	0.586	0.674	0.903	0.792	0.844
60%	0.874	0.552	0.677	0.886	0.599	0.715	0.784	0.601	0.681	0.906	0.793	0.846
70%	0.875	0.577	0.696	0.901	0.611	0.728	0.789	0.608	0.687	0.905	0.794	0.846
80%	0.876	0.576	0.695	0.906	0.628	0.742	0.790	0.620	0.695	0.901	0.799	0.847
90%	0.887	0.608	0.722	0.906	0.628	0.742	0.798	0.630	0.704	0.907	0.806	0.853

TP	ArgoUML 0.10.1 to 0.12 (110775 instances, 15.2%)			ArgoUML 0.12 to 0.16 (35476 instances, 16.8%)			ArgoUML 0.16 to 0.181 (113171 instances, 14.5%)			ArgoUML 0.181 to 0.22 (22442 instances, 17.6%)		
	Precision	Recall	F-M	Precision	Recall	F-M	Precision	Recall	F-M	Precision	Recall	F-M
10%	0.723	0.435	0.543	0.635	0.477	0.545	0.652	0.465	0.543	0.776	0.620	0.689
20%	0.779	0.477	0.591	0.735	0.480	0.581	0.686	0.502	0.580	0.796	0.619	0.697
30%	0.795	0.503	0.616	0.750	0.507	0.605	0.718	0.518	0.602	0.805	0.605	0.691
40%	0.801	0.516	0.628	0.761	0.517	0.616	0.731	0.532	0.616	0.832	0.593	0.693
50%	0.800	0.537	0.642	0.765	0.528	0.625	0.726	0.539	0.619	0.826	0.597	0.693
60%	0.806	0.547	0.652	0.755	0.530	0.623	0.730	0.542	0.622	0.834	0.600	0.698
70%	0.798	0.553	0.654	0.764	0.549	0.638	0.731	0.548	0.627	0.830	0.607	0.701
80%	0.803	0.570	0.667	0.764	0.544	0.636	0.736	0.552	0.631	0.825	0.626	0.712
90%	0.803	0.576	0.671	0.770	0.555	0.645	0.743	0.554	0.634	0.835	0.607	0.703

^aThe percentage number after instances denotes the prior probability for *yes*

^bTP denotes *Training Percentage*

^cF-M denotes *F-Measure*

Table 4. Results for different Releases of Azureus2 and ArgoUML

cally other changes of the same transaction. By incorporating static source code dependencies, we are able overcome this lack of information on the *causality between changes* at least partially. Moreover, due to the robustness against uncertainty, a Bayesian network-based approach performs well under these conditions.

5.1. Lessons learned

In the phase of software maintenance or reverse engineering, given a revision request that affects a software entity, all the other entities are possible candidates for a co-change. However, whether two entities change together depends on numerous factors.

From a software engineering perspective, *good designs* have the characteristics of high cohesion and low coupling among the modules brought by information hiding and separation of concerns [21] which implies that ideally, changes to one entity should entail changes to no or at least to only a small set of other entities. Therefore it is neither nec-

essary, nor realistic—due to the size of modern software systems—to consider all other entities as candidates; by leveraging traditional dependency and change significance analysis with software evolution techniques, such as change coupling detection, we can narrow down the amount of data to take into consideration and provide better tools for software engineers.

5.2. Threats to Validity

The first group of potential threats to validity emerges from the quality of the underlying data and of limitations of the tools and algorithms used to extract the features that are need to train the Bayesian network. For example static source code dependency information is an important input to our approach. The compilation level and the availability of all the third-party libraries influence the quality of the output of our FAMIX transformation tools. Furthermore the algorithm for recovering the transactions from the CVS system is based on heuristics and therefore not guaranteed

to produce optimal results.

The other group of potential threats is concerned with the fact that there is partially subjectivity involved in our approach: First, we have selected a set of features for training the Bayesian network that seemed reasonable for us. Second, in the process of discretizing the dependency weights and change significance levels into different classes, we relied on experiences from past research on similar projects.

6. Related Work

Throughout this paper we presented our Bayesian network based approach with a set of selected features to predict change couplings. Many other methods have been developed in this field. In [14], Kagdi and Collard provided a taxonomy of the approaches for mining software repositories (MSR). Our work fits in the category of *MSR via Data Mining*. In this section, we will review related work mainly in this class. Basically, the proposed approaches can be identified with two key points: data source, and mining techniques.

Ying *et al.* [29] developed an approach that applies association rules to co-changed files. The hypothesis is that past co-changed files can be used to recommend potentially relevant source code changes to a change request. Association rule algorithms extract frequently co-changed entities of a transaction into sets which are regarded as change patterns to guide future change couplings. Since this approach uses co-change history in CVS, it avoids source code dependency parsing process. Therefore it does not limit to predict source file co-change. However, the association rule algorithm just calculates the frequency co-changed items in the past, it omits many other information and does not help to reflect the cause-effect relation hidden in this change coupling phenomenon. Besides, ‘frequently co-changed entities’ is a subjective variable, sometimes hard to define in different projects. We believe that the reported performance with around 20% to 30% recall rate suffered from the loss of information.

Zimmerman *et al.* [30], similar to the work of Ying, used co-change history and association rule algorithm to predict change coupling. Although there is a slight difference in the form of the algorithm, they share the common sets of characteristics of this type of approach. And the performance of both work are at the similar quantitative level.

Knab and Pinzger [15] applied a decision tree-based algorithm to predict defect densities in source code files. They extracted modifications, defect report metrics, number of incoming and outgoing calls from source releases and version history database. The attempt to focus more on the understanding of the factors that lead to defects is similar to our motivation of using Bayesian network.

Mirarab *et al.* [20], independently from us, noticed the

importance of Bayesian network to model the uncertainty of the change coupling process and proposed a Bayesian network based approach to predict change propagation. They proposed three Bayesian network models, *i.e.*, Bayesian Dependency Model (BDM), Bayesian Dependency and History Model (BDHM), and Bayesian History Model (BHM). The first model relied only on the the dependency information; the second incorporated the co-change history information; the last relied only on the co-change history. The average performance of BDHM is the best among these three model evaluated in the case study of Azureus. The idea of BDHM is similar to ours. However, the authors did not consider any other features which we argued to be important factors for change coupling prediction. Our experiment on the same case study reveals the outperformance of our approach, especially with the improvement on the recall rate.

Ratzinger *et al.* [26], investigated different kinds of data mining techniques on the features, such as growth measures, relationships between classes, authors information etc., extracted from versioning systems, and compared the prediction results for future refactoring locations. Although the prediction target is different, the consideration of incorporating different factors besides co-change history and code dependency is similar to ours.

Besides the work from the *MSR via Data Mining* category, there are many approaches developed from other perspectives. Hassan *et al.* [13] introduced several heuristics to be used to predict change propagation by suggesting entities that could possibly co-change. This approach chose the candidates from the heuristic data source first and pruned them. The average prediction performance (precision and recall rate) of the hybrid heuristics on the five studied software systems is around 50%. Gall and Jazayeri [11] applied relation analysis technique to detect similar change patterns in different parts of systems (a.k.a., logical couplings). Though this technique is useful to identify even architectural weakness, it does not predict change couplings.

7. Conclusion and Future Work

In this paper, we presented an approach based on the idea of Bayesian networks to predict the change coupling behavior between source code entities. We extract a set of features, *i.e.*, static source code dependency, past co-change frequency, change significance level, age of change, author information, change request, change candidate, and co-changed entity. The Bayesian network models the uncertainty in the change coupling process based on these features. We conducted experiments on two medium open source systems, *i.e.*, Azureus and ArgoUML. The results of the case study support our contributions: first, given a change request, we showed that the set of features we chose

to build the Bayesian network provides useful predictions about the change coupling candidates. Second, compared to the results of previous work on Bayesian network models in the context of change prediction, we achieve a better performance in terms of precision and recall. Our approach shows a precision rate of 60% and a recall rate of 40% already after a few training phases. These rates increase up to 80% for precision and 60% for recall once the initial training has been completed.

For instance, our approach can be used to extend or augment existing tools in the domain of version control systems or integrated development environments in order to provide recommendations of the kind *"If you change this package, then you will probably also have to consider changing these packages..."* to the developer.

Possibilities for future work are the investigation whether additional features and other combinations of features will give better prediction results. Such new sets of features can be validated by means of more case studies. Of special interest are industrial projects that have different properties, e.g., strict code ownership, and how they impact our results.

8. Acknowledgements

Sincere thanks are given to Beat Fluri, Martin Pinzger and Jiwen Li for their suggestions in improving the quality of the paper. Special thanks are given to Siavash Mirarab for his sharing of the release information with us on the case study. This work was supported by the Hasler Foundation as part of the ProMedServices project.

References

- [1] K. Bennett and V. Rajlich. Software maintenance and evolution: a roadmap. *Proceedings of the conference on The future of Software engineering*, pages 73–87, 2000.
- [2] D. Chickering. Learning Bayesian Networks is NP-Complete. *Learning from Data: Artificial Intelligence and Statistics V*, 1996.
- [3] G. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9(4):309–347, 1992.
- [4] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0-the FAMOOS information exchange model. URL: <http://www.iam.unibe.ch/famoos/FAMIX>, 9, 1999.
- [5] N. Fenton, P. Krause, and M. Neil. Software measurement: uncertainty and causal modeling. *Software, IEEE*, 19(4):116–122, 2002.
- [6] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, 1999.
- [7] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from version control and bug tracking systems. *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32, 2003.
- [8] B. Fluri and H. Gall. Classifying Change Types for Qualifying Change Couplings. *Proceedings of the 14th International Conference on Program Comprehension (ICPC)*, pages 35–45, 2006.
- [9] B. Fluri, H. Gall, and M. Pinzger. Fine-Grained Analysis of Change Couplings. *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 66–74, 2005.
- [10] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.
- [11] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 13–23, 2003.
- [12] T. Girba, S. Ducasse, and M. Lanza. Yesterday's Weather: guiding early reverse engineering efforts by summarizing the evolution of changes. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 40–49, 2004.
- [13] A. Hassan and R. Holt. Predicting change propagation in software systems. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 284–293, 2004.
- [14] H. Kagdi, M. Collard, and J. Maletic. Towards a taxonomy of approaches for mining of source code repositories. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.
- [15] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. *Proceedings of the 2006 international workshop on Mining software repositories*, pages 119–125, 2006.
- [16] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. Codecrawler: an information visualization tool for program comprehension. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 672–673, New York, NY, USA, 2005. ACM.
- [17] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution-the nineties view. *Proc. Metrics*, 97:5–7, 1997.
- [18] M. Madden. The Performance of Bayesian Network Classifiers Constructed using Different Techniques. *Proceedings of European Conference on Machine Learning, Workshop on Probabilistic Graphical Models for Classification*, pages 59–60, 2003.
- [19] T. Mens and S. Demeyer, editors. *Software Evolution*. Springer-Verlag Berlin Heidelberg, 2008.
- [20] S. Mirarab, A. Hassouna, and L. Tahvildari. Using Bayesian Belief Networks to Predict Change Propagation in Software Systems. *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 177–188, 2007.
- [21] D. Parnas. On the criteria to be used in decomposing systems in modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [22] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
- [23] M. Pinzger. ArchView-Analyzing Evolutionary Aspects of Complex Software Systems. *Vienna University of Technology*, 2005.
- [24] M. Pinzger, K. Graefenhain, P. Knab, and H. Gall. A tool for visual understanding of source code dependencies. *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2008.
- [25] R. Purushothaman and D. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.
- [26] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining Software Evolution to Predict Refactoring. *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 354–363, 2007.
- [27] I. Stamelos, L. Angelis, P. Dimou, and E. Sakellaris. On the use of Bayesian belief networks for the prediction of software productivity. *Information and Software Technology*, 45(1):51–60, 2003.
- [28] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.
- [29] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.
- [30] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.