Behavioral Dependency Measurement for Change-proneness Prediction in UML 2.0 Design Models

Ah-Rim Han, Sang-Uk Jeon, Doo-Hwan Bae
Department of Computer Science, Korea Advanced Institute of Science and Technology
{arhan, sujeon, bae}@se.kaist.ac.kr

Jang-Eui Hong
School of Electrical and Computer Engineering, Chungbuk National University
jehong@chungbuk.ac.kr

Abstract

During the development and maintenance of Object-Oriented (OO) software, the information on the classes which are more prone to be changed is very useful. Developers and maintainers can make a more flexible software by modifying the part of classes which are sensitive to changes. Traditionally, most change-proneness prediction has been studied based on source codes. However, change-proneness prediction in the early phase of software development can provide an easier way for developing a stable software by modifying the current design or choosing alternative designs before implementation. To address this need, we present a systematic method for calculating the Behavioral Dependency Measure (BDM) which helps to predict change-proneness in UML 2.0 models. The proposed measure has been evaluated on a multi-version medium size open-source project namely JFreeChart. The obtained results show that the BDM is an useful indicator and can be complementary to existing OO metrics for change-proneness prediction.

1. Introduction

Software engineering deals with the construction of multiversion software, that is, software that will undergo a number of changes either to enhance the functionality or to fix bugs [13]. Some parts of software may be more prone to changes than others. The likelihood of changes to be occurred is referred as *change-proneness*. Predicting change-proneness can be very helpful in software development phases because it indicates the design quality of the software. If the modification of a class method exhibits a large sensitivity to changes, the corresponding design has a quality problem and needs to be improved by redesigning. In addition, change-proneness prediction aids to allocate resources of inspection and testing efficiently.

There have been several research efforts for predicting change-prone classes in OO software based on source codes. Code-based change-proneness prediction is used at the maintenance phase. However, if the change-prone classes can be predicted at the earlier phase in the software development life cycle, when the design models become available, design quality problems can be detected before implementing codes; we can modify the current design or choose alternative designs easily on design models. Thus, model-based change-proneness gives a high return on investment partially because decisions made on design models have substantial downstream consequences and checking and fixing design models are relatively inexpensive. This is important since the largest percentage of software development effort is spent on maintenance [12]. As a result, model-based change-proneness prediction would contribute to improve software quality and save development cost.

The main challenge of model-based change-proneness prediction is to develop a method that can predict change-prone classes effectively within limited information compared to source codes. To predict change-proneness from design models, we provide the Behavioral Dependency Measure (BDM) based on the interactions between objects. This measure can be used for ranking classes according to the proneness that they are likely to be changed. We first define behavioral dependencies which might cause changes. Then, we describe the steps for calculating the BDM based on the defined behavioral dependencies using UML 2.0 design models. The used diagrams are a Class Diagram (CD), Sequence Diagrams (SD), and an Interaction Overview Diagram (IOD). The SD and IOD provide a scenario-based behavior modeling. *Alt, loop* combined fragments used in a SD enable to model complex control structures same as in source codes. The IOD, newly introduced in UML 2.0, represents an overview of the control flow of the complete software. The BDM is evaluated on a multi-version medium size open-source project namely JFreeChart [2]. To investigate the effectiveness of the BDM for predicting change-proneness, we reversed JFreeChart source codes to UML design models and measured the BDM based on them. Then, to measure the extent of changes, we extracted the modified lines of code of each class in the next six releases. We regressed these changes on the BDM and other existing OO metrics to prove the usefulness of the BDM.

The rest of the paper is organized as follows: Section 2 defines the behavioral dependencies which might cause changes and discusses some issues for accurate change-proneness prediction. Section 3 describes how to calculate the BDM in a systematic way. Section 4 presents a case study as an evaluation of the proposed BDM. Section 5 describes related researches. Section 6 concludes and outlines future research.

2. Change Prediction

2.1. Behavioral Dependency Definitions

Changes in a class occur in two ways: by modifications to a class itself and by changes propagated from other classes. The changes which occur by the former, which we call internal changes, refer to addition/deletion of attributes, modification to method declarations, etc. However, they are difficult to be predicted using design models; data such as source lines of code (SLOC), number of fields (NOF), and number of parameters (NOP) can be obtained from source codes. Therefore, we only consider changes that occur by the latter, which we call propagated changes. Such changes can be predicted by examining dependencies of pairs of entities (i.e., classes or objects) in the system. Among several types of dependencies, we use a *behavioral* dependency. We assume that the object sending a message has a behavioral dependency to the object receiving the message. This is from the insight that modifying the object receiving the message may affect the object sending the message. We also assume that a high intensity of a behavioral dependency represents high possibility of changes to be occurred. The rationale behind this assumption is that the more external services an object is dependent on, the more likely it is that the class of which the object is an instance will be changed. To quantify the behavioral dependency, we define two kinds of behavioral dependencies: direct and indirect. Each of them is defined as follows. Let a system consists of the objects op_1, op_2, ..., op_q, ..., op_r, ..., op_n.

Definition 1 (Direct behavioral dependency). The object op_1 has a direct behavioral dependency to the object op_2 if op_1 needs some services of op_2 by sending a synchronous message to op_2 and receiving a reply from op_2.

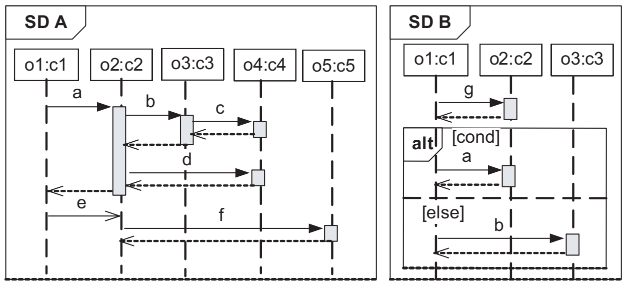


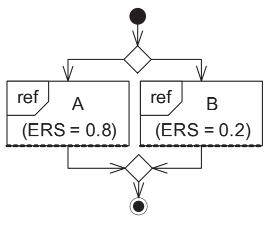
Figure 1: An example of sequence diagrams

Definition 2 (Indirect behavioral dependency). The object op_1 has an indirect behavioral dependency to the object op_r when the following conditions are met; op_1 needs some services of the object op_2 by sending a synchronous message to op_2, and subsequently op_2 needs services of the object op_q before replying to op_1. op_q may be op_r or may need some services of other objects before replying to op_2, and finally the service of op_r is needed.

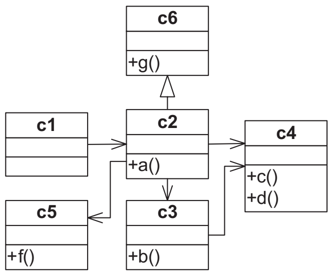
A synchronous message entails a dependency between two objects since the sender object depends on the receiver object. On the other hand, an asynchronous message does not entail such dependency since the sender object does not wait for a reply but continues to proceed. Therefore, in our approach, we only regard synchronous messages with replies.

Figure 1 shows two example SDs. In SD A, the object o1 has a direct behavioral dependency to the object o2 by sending a synchronous message a to o2 and receiving a reply from it. On the other hand, the object o1 has an indirect behavioral dependency to the object o3; before the object o1 receives a reply for the message a from the object o2, the message b is sent from the object o2 to the object o3. The asynchronous message e from the object o1 to the object o2 does not entail a behavioral dependency since the sender object o1 does not wait for a reply from the object o2. All the message in SD B incur direct behavioral dependencies.

Note that, an indirect behavioral dependency is not a transitive relation. For example in Figure 1, the object o1 and o5 do not have a behavioral dependency, even though the objects o1 and o2 have a behavioral dependency by the message a and the object o2 and o5 have a behavioral dependency by the message f. This is because the message f is sent from o2 to o5 after o1 receives the reply of the message a from o2. To identify the indirect behavioral dependency between two objects precisely, we need to save the information of the message which triggers the current message. Hence, when o1 has an indirect dependency to o2, we can derive a reachable path which is a sequence of



(a)



(b)

Figure 2: (a) An example of the interaction overview diagram. (b) An example of the class diagram which is consistent with SDs in Figure 1.

exchanged messages between the two objects by traversing stored messages from o2 to o1 in a backward direction.

2.2. Considerations for accurate prediction

There have been several researches for predicting change-prone classes using established metrics of OO software. However, for more accurate and consistent change-proneness prediction, more considerations are needed. In this paper, we consider the following issues.

First, established couplings based on method call dependencies do not take account of the extent to which one class is dependent on another class. In other words, no matter how many times the class c1 calls the method of the class c2, the established couplings of static aspect treat this as one. Let us consider two cases; the class c2 implements one method called by the class c1 for 100 times while the class c4 implements two methods called by the class c1 for once of each method. The established static couplings for the former case and the latter case are one and two, respectively. However, the class c1 might be more behaviorally dependent on the class c2 than the class c4. Hence, we consider the dynamic aspect of coupling between objects by taking not only the number of messages, but also the execution rate of the messages based on the control structure and the operational profile. A SD in UML 2.0 provides combined fragments to express control structures such as branch and loop. A message in an *alt* combined fragment which corresponds to a branch control structure can be executed depending on the condition. This may affect the behavioral dependency of the objects which are related by this message. Without running a program (i.e., dynamic information), it is not feasible to determine whether the message will be executed or not. Therefore, the probabilistic execution rate of a message is considered when measuring a behavioral dependency. For example, in the SD B of Figure 1, either the message a or the message b is executed whether the condition is satisfied or not (i.e., true or false). Therefore, the probabilistic execution rate of each message can be 0.5. The IOD in UML 2.0 illustrates an overview of a flow of control in which each activity node can be a SD. Some scenarios (i.e., SDs) might be executed more frequently than others, as specified in an operational profile [10]. The operational profile provides the expected execution rate of a SD. Therefore, it also needs to be considered for measuring the better behavioral dependency. We suggest to specify the Expected Execution Rate (ERS) (i.e., operational profile) of each SD in an IOD. For example, consider the IOD in Figure 2(a). It shows that the expected execution rates of SD A and SD B are 80% and 20%, respectively.

Second, due to inheritance, when an object sends a message to the other object, the class of the receiver object may be different from the class implementing the corresponding method. For example, in the SD B in Figure 1 and the CD in Figure 2(b), the object $o1$ sends the message g to the object $o2$. However, the actual implementation of the method g is in the class $c6$ which is different from the class of the object $o2$. Hence, the change of the method g in the class $c6$ is propagated to the class $c1$. Without considering the inheritance, the behavioral dependency would be misidentified. For this reason, when a sender object sends a message to a receiver object, we check whether the method of the message is implemented in the class of the receiver object or in one of its ancestor classes.

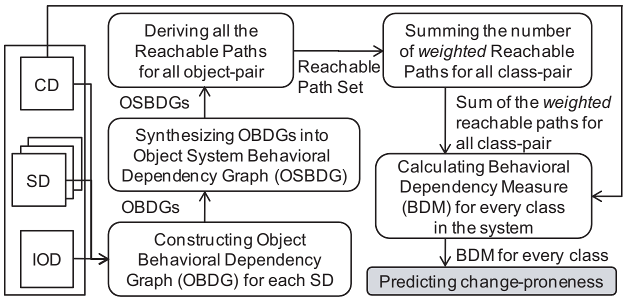

CD
SD
IOD
Deriving all the Reachable Paths for all object-pair
Reachable Path Set
Summing the number of *weighted* Reachable Paths for all class-pair
OSBDGs
Synthesizing OBDGs into Object System Behavioral Dependency Graph (OSBDG)
Sum of the *weighted* reachable paths for all class-pair
OBDGs
Calculating Behavioral Dependency Measure (BDM) for every class in the system
Constructing Object Behavioral Dependency Graph (OBDG) for each SD
BDM for every class
Predicting change-proneness

Figure 3: Overview of our approach for predicting change-proneness.

3. Behavioral Dependency Measurement

In this section, we provide a systematic way for calculating the BDM in UML design models which are SD, CD, and IOD. An overview of our approach is depicted in Figure 3. BDM is computed from the following procedures. Object Behavioral Dependency Graph (OBDG) is constructed for each SD based on all direct and indirect behavioral dependencies between objects by referring the CD and the IOD. After that, we synthesize all OBDGs into Object System Behavioral Dependency Graph (OSBDG) for the whole system. Next, we derive all the reachable paths for each pair of objects in the system from the OSBDG. A reachable path is weighted by a distance between objects and an execution rate of the messages composing of this reachable path. Then, we sum a number of weighted reachable paths for all pairs of classes. Finally, we calculate BDM for every class in the system. Detailed procedures are described in the following subsections.

3.1. Constructing OBDG and OSBDG

A dependency graph $OBDG_A$ for a SD A is a 2-tuple $\{O, M\}$, where

- O is a set of nodes representing objects in the SD
- M is a set of edges representing messages that are exchanged between two nodes. A message $m \in M$ represents a synchronous message with a reply, which entails a direct dependency from a sender object to a receiver object. It has following 6 attributes.
  - $m_s \in O$ is the sender of the message.
  - $m_r \in O$ is the receiver of the message. $m_r \neq m_s$
  - m_n is the name of the message.
  - $m_b \in M$ is the instance of a backward navigable message. $m_b \neq m$. "-" means none.
  - m_{meL} is the probabilistic message execution rate in a sequence diagram. $0 \leq m_{meL} \leq 1$. Default value is 1.
  - m_{meH} is the expected message execution rate in an interaction overview diagram. $0 \leq m_{meH} \leq 1$. Default value is 1.

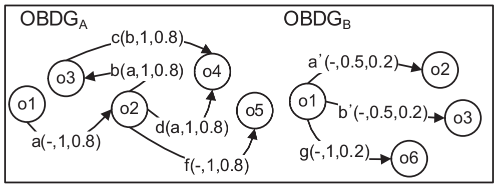
m_s and m_r represent the sender and receiver objects, respectively. Since we do not consider messages from an object to itself, they should not represent the same node. As we pointed in Section 2.2, the class of the object receiving a message may be different from the class implementing the corresponding method. In other words, the object may send the message which is the method of the superclass of the class of the object receiving the message. In such case, the object sending a message may change when the actual implemented method changes. Therefore, when binding a receiver node, we make sure whether the method is implemented in the actual class of the receiver object. If not, the receiver node of the message is bound to an object of an ancestor class which actually implements the method.

m_b represents the message that triggers m, and is called as the backward navigable message. By tracing the backward navigable message, we can identify the message that activates the current message. As we noted in Section 2.2, m_b is essential for identifying indirect behavioral dependencies between objects. m_b also prevents an infinite loop problem when deriving a reachable path from the OSBDG.

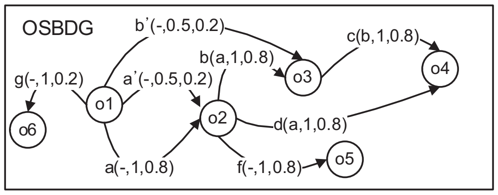
m_{meL} and m_{meH} help to predict change-proneness better by considering the probabilistic or expected execution rates of the messages as mentioned in Section 2.2. Later, these rates are synthesized according to a reachable path to be used for measuring a behavioral dependency. m_{meL} represents the probabilistic message execution rate in a SD. We consider a branch control structure which might affect the probability of the message execution. When a message is in an *alt* combined fragment, it is executed only when a condition of the corresponding interaction operand is met. Therefore, m_{meL} is the same as the probability that one of the interaction operands is selected. If an *alt* combined fragment is nested, the probability of message execution in the corresponding combined fragment is multiplied to m_{meL} recursively. When a message is not contained in any combined fragments, m_{meL} is 1. m_{meH} represents the expected message execution rate in an IOD. We specify the ERS (i.e., operational profile) of each SD in an IOD. A message in a SD is executed only when the corresponding SD is activated. Therefore, m_{meH} is the same as the probability that the control flow of the software is reached to the SD to which the message belongs. This can be obtained by multiplying all the values of ERS on the way from the initial node to the corresponding SD node in the IOD. If a SD is always to be activated, m_{meH} of all the messages in the SD are 1.

Figure 4(a) shows two OBDGs, each of which corresponds to the SD A and the SD B, respectively, in Figure 1. The ERS value of each SD is presented in Figure 2(a). Figure 2(b) illustrates the CD of objects showed in these SDs. Each message is represented as the form of $m_n(m_b, m_{meL}, m_{meH})$. In the $OBDG_B$, to distinguish the messages from those in the $OBDG_A$, we renamed the message a to a' and b to b'. Since either the message a' or b' may be activated depending on the condition of the *alt* combined fragment, the a'_{meL} and b'_{meL} are 0.5, respectively. The ERS values of the SD A and the SD B are reflected into the messages' execution rates. Note that, the receiver node of the message g is $o6$, since $c6$ implements the method g.

To determine the behavioral dependencies between objects in the whole system, we synthesize OBDGs into the $OSBDG = \{O_s, M_s\}$. O_s and M_s denote the set of objects and the union of messages existed in the system, respectively. The method for constructing the OSBDG will be explained using the example in Figure 4. 4(b) shows the OSBDG by synthesizing two OBDGs in Figure 4(a). This OSBDG is composed of $O_s = \{o1, o2, .., o6\}$ and $M_s = \{(M \text{ of SD A}) \cup (M \text{ of SD B})\}$. Note that, the object $o1$ in SD A and the object $o1$ in SD B are instantiated from same class $c1$, therefore only one $o1$ is remained in the OSBDG. The sending message a from $o1$ in SD A and the other sending message a' from $o1$ in SD B are connected



(a)



(b)

Figure 4: (a) OBDG$_A$ and OBDG$_B$ correspond to SD A and SD B in Figure 1. (b) OSBDG after synthesizing two OBDGs in (a).

with the corresponding target object $o2$ in the OSBDG. If the message m is triggered by the other message in the context of the system by examining the IOD, we set this other message as a backward navigable message of the message m. There is no such case in this example.

3.2. Deriving Reachable Paths

We derive reachable paths for all pairs of objects in the system using OSBDG. To retrieve the reachable paths from a source object $o1$ to a target object $o2$, we start traversing of the OSBDG from an incoming message of $o2$ to an outgoing message of $o1$ in a back direction. When $o1$ has a direct behavioral dependency to $o2$, one of the incoming messages of $o2$ and one of the outgoing messages of $o1$ are equal so that this message is added into a set of reachable paths. On the other hand, when $o1$ has an indirect behavioral dependency to $o2$, we traverse the OSBDG from one of the incoming messages of $o2$ iteratively by substituting a backward navigable message for it and finally reach to one of the outgoing messages of $o1$. A stored sequence of messages during traversing is a reachable path. An example of the reachable path set from $o1$ to $o3$ in Figure 4(b) is $\{$*ab,b'*$\}$. The method for retrieving reachable paths from an object $o1$ to the object $o2$ is presented in Algorithm 1.

Algorithm 1 ReachablePathSet(o1:O, o2:O)

```
input $OUT \leftarrow$ outgoing message set of $o1$
input $IN \leftarrow$ incoming message set of $o2$
input $RP \leftarrow \emptyset$ an array for storing reachable path
input $RPS \leftarrow \emptyset$ a vector for saving a set of reachable paths
output $RPS$

for all $in \in IN$ do
  for all $out \in OUT$ do
    if $in == out$ then
      /*For RPS by the Direct Behavioral Dependency*/
      $RPS \leftarrow RPS \cup \{in\}$
    else
      /*For RPS by the Indirect Behavioral Dependency*/
      $RP \leftarrow RP + \{in\}$
      while $in_b != out$ && $in_b != \emptyset$ do
        if $in == out$ then
          $RP \leftarrow RP + \{in_b\}$
          $RPS \leftarrow RPS \cup RP$
          $RP \leftarrow \emptyset$
          break
        else
          $RP \leftarrow RP + \{in_b\}$
          $in \leftarrow in_b$
```

3.3. Calculating Behavioral Dependency Measure

Prior to calculate BDM for every class in the system, we sum the number of weighted reachable paths for all pair of classes. Let RPS be a set of all reachable paths in the system and s be one reachable path in the set. f denotes the first message in the reachable path s, in other words, the outgoing message from the sender object. We formalize the sum of the number of weighted reachable paths (WRP) from class c1 to class c2 as follows.

$$SumWRP(c1, c2) = \sum_{\forall s \in RPS(o_1, o_2)} DF(s) \times f_{meH} \times f_{meL}$$

We use three items for weighting a reachable path: DF, m_{meH} and m_{meL}. We denote a distance factor by $DF(s) = 1/d$, where d is the distance length which is the number of messages in the corresponding reachable path s. The rationale for using the distance factor is that an indirect behavioral dependency might be weaken as it is occurred by the successive calls. In other words, the farther an object is from the source of changes, the less the object is likely to be changed. Therefore, we need to degrade the impact when two objects have a long distance of indirect behavioral dependency.

Finally, BDM for every class in the system is obtained as follows. Let C denote all the classes in the system.

$$BDM(c1) = \sum_{\forall c_n \in C} SumWRP(c1, c_n)$$

Table 1 summarizes the sum of the number of weighted reachable paths obtained from the OSBDG in Figure 4(b) and BDM of each class. BDM is used in predicting change-proneness; the higher BDM the class has, the larger the likelihood to be changed.

Table 1: Sum of the number of weighted reachable paths of all class-pair and BDM of each class (Row: Receiver, Column: Sender)

	$c1$	$c2$	$c3$	$c4$	$c5$	$c6$	$BDM(c)$
$c1$	0	0.81	0.41	0.67	0	0.2	2.09
$c2$	0	0	0.8	1.2	0.8	0	2.8
$c3$	0	0	0	0.8	0	0	0.8
$c4$	0	0	0	0	0	0	0
$c5$	0	0	0	0	0	0	0
$c6$	0	0	0	0	0	0	0

4. Case Study

This section presents the results of a case study whose objectives are to provide an empirical validation of the BDM presented above. The first subsection explains in more detail of the studied system, the experimental goal, and the method we follow. In the next subsection, results are presented and interpreted.

4.1. Experiment Design

To calculate the BDM which is measured on UML models, we developed a tool, BADAMO (BehAvioral Dependency Analyzer of UML MOdels). The BADAMO is built on EMF (Eclipse Modeling Framework), and imports the UML 2.0 models in the format of XMI generated from Rational Software Architect (RSA) 7.0 which is Eclipse-based UML 2.0 modeling tool by the Rational Division of IBM. In order to investigate whether the BDM is statistically related to change-proneness, we need the case system which has well-documented UML models and subsequent releases for extracting change related information. However, in practice, most legacy systems which had been developed and maintained for a long time do not have design documents. Therefore, to perform our experiment, we reversed the existing system, JFreeChart[2](version 1.0.0) to the UML design models using a reverse engineering tool with manual supports. JFreeChart is an open-source Java class library for generating various types of charts. It has been developed for about seven years. There are 43 releases (at the time we conducted this case study) between the first version 0.5.6, released on 1 December 2000, and the last version 1.0.6, released on 15 June 2007. When reversing the UML models from the JFreeChart codes, we only extracted the information which is needed for calculating the BDM. We excluded classes which have behavioral dependencies from/to the library, since the scope of the measurement is limited to the application classes of JFreeChart. In addition, internal messages (i.e., method call within same class) were not considered, since they do not occur behavioral dependencies. A SD was simply constructed based on consecutive synchronous calls, which corresponds to the notion of the reachable path. Figure 5 shows an example of the reversed sequence diagram. This SD corresponds to the reachable path from the object of the class $WaterMapPlot$ to the object of the class $DefaultKeyedValues$ with three messages. We extracted 5659 reachable paths by analyzing the source codes of JFreeChart, and finally obtained 5659 SDs with 12563 messages. The IOD can not be reversed from source codes; it is specified only from the early stage of the development to help developers get an overview of the system. Thus, this information is not reflected when implementing codes. Hence, in this experiment, the ERS in an IOD was not considered when calculating the BDM. The CD was built automatically using RSA 7.0.

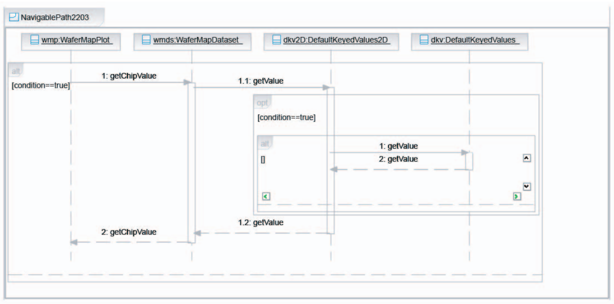


Figure 5: A sequence diagram reversed from source codes of the JFreeChart version 1.0.0

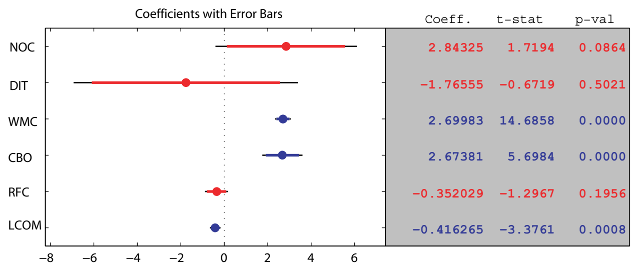
Stepwise multiple regression was used to build the change-proneness prediction models. The dependent variable of the model is change-proneness. The change-proneness in this study is the total amount of changes (source lines of code added and deleted) across the six releases from version JFreeChart 1.0.0 to version JFreeChart 1.0.6. Change data was collected for each application class. Since it is not feasible to predict the change-proneness of the newly created class, we only consider changes of the 369 classes which have been existed from version JFreeChart 1.0.0. Based on the change data, we used the Source Code Counter [1] to compute the amount of changes of each class. The independent variables include the Chidamber and Kemerer (C&K) metrics [9] and the BDM. C&K metrics are most widely used metrics for evaluating an object-oriented software. We choose the 6 metrics that are available on UML models from C&K metrics: NOC (number of children), DIT (depth of inheritance tree), WMC (weighted methods per class), RFC (response for a class), CBO (coupling between objects), and LCOM (lack of cohesion in methods). The independent variables are measured on version JFreeChart 1.0.0 that is the first release of the studied system.

A primary goal of this case study is to investigate whether the BDM is an useful indicator and is additional and complementary to existing C&K metrics for explaining variance of change-proneness. To achieve this goal, we proceeded in two steps. First, we carried out stepwise multiple regressions using C&K metrics in order to generate a multivariate regression model that would serve as a baseline of comparison. We then continued by performing stepwise multiple regression, using C&K metrics, and the BDM. If the goodness of fit of the latter model turns out to be significantly better than the former model, we would then be able to conclude that the BDM is the useful and additional explanatory variable for change-proneness prediction.

4.2. Results

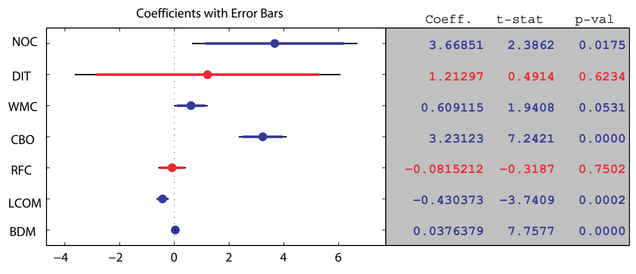
The first result of the stepwise regression when using C&K metrics as candidate covariates is presented in Figure 6(a). After removing 7 outliers that are clearly over-influential on the regression results (with an extremely large change value), we obtained the prediction model with 3 variables included. The model explains around 56 percent of the variance of the data set and shows an adjusted R^2 of 0.55. WMC, CBO and LCOM were the first, second and last variables to be included, respectively. When including BDM in addition to C&K metrics to make a prediction model, we obtain the result as in Figure 6(b). Around 64 percent of the variance in the data set is explained and an adjusted R^2 of 0.64 is obtained. In this latter model, WMC, BDM, CBO, LCOM, NOC variables were included in the order of the sequence as listed. Note that, BDM was the second variable to be included with significant-level of p-value < 0.00001. Therefore, even when accounting for the difference in the number of covariates, the coefficient of determination (R^2) is increased by 9 percent or 20 percent of the unexplained variance (from 0.55 to 0.64) by using BDM. This results support the usefulness of BDM which is a complementary indicator to C&K metrics as far as change-proneness is concerned. In other words, this shows the importance of considering the behavioral dependency of the system which uses the inheritance and polymorphism to predict change-proneness.

The fitness of the models for the model-based change-proneness prediction is rather low compared to code-based change-proneness prediction, since the information extracted from UML models is not enough as from source codes. If other metrics which are derivable from only source codes are considered when building the change-proneness prediction model, the R^2 values will be higher. For example, LOC which indicates the size of the class is known as a significant indicator for affecting change-proneness [3]. Recall that the goal of this study is to determine whether BDM helps to obtain a better model fit, therefore, provides the improved predictive model compared to the model considering only C&K metrics which are available on UML models. This provides the benefit of the early change-proneness prediction, when a design model becomes available, without implementing the codes.



	Coeff.	t-stat	p-val
NOC	2.84325	1.7194	0.0864
DIT	−1.76555	−0.6719	0.5021
WMC	2.69983	14.6858	0.0000
CBO	2.67381	5.6984	0.0000
RFC	−0.352029	−1.2967	0.1956
LCOM	−0.416265	−3.3761	0.0008

(a)



	Coeff.	t-stat	p-val
NOC	3.66851	2.3862	0.0175
DIT	1.21297	0.4914	0.6234
WMC	0.609115	1.9408	0.0531
CBO	3.23123	7.2421	0.0000
RFC	−0.0815212	−0.3187	0.7502
LCOM	−0.430373	−3.7409	0.0002
BDM	0.0376379	7.7577	0.0000

(b)

Figure 6: Results of stepwise regression using (a) C&K metrics, (b) C&K metrics and BDM.

5. Related Work

Previous attempts to predict change-proneness classes are associated with the studies for assessing the external quality factors such as maintainability, flexibility, changeability, and stability of OO software [4, 7, 12, 8]. However, model-based change-proneness prediction is rarely discussed whereas there have been rich algorithms and tools for source code level prediction [14]. Arnold and Bohner [5] give an overview of several formal models of change propagation. They introduce several tools and techniques that are based on code dependencies and algorithms such as slicing and transitive closure. In [11], a set of algorithms that determine what classes are affected by a given change is proposed. The methodology represents a system as a set of data dependency graphs.

There are studies that investigate the relationship between existing design metrics such as coupling measures and the impact of changes. Briand et al. [6] empirically investigated whether coupling measures are related to ripple effects using a commercial OO system. The aim is to rank classes according to their probabilities of containing ripple effects when a change is occurred on one class, while our approach aims at quantifying the degree of dependencies between classes to predict the change-proneness of classes in a future generation. In addition, traditional coupling measures fail to capture complex dependencies due to the inheritance and the polymorphism. This results in poorer precision of change-proneness prediction for the software that uses inheritance and polymorphism to improve internal reuse in a system. In our work, we take account of the actual call dependency between objects by finding the class which implements the method being called. Arisholm et al. [3] investigates the use of dynamic coupling measures as indicators of change-proneness. Their approach is based on correlating the number of changes, a continuous variable which represents change-proneness, to each class with dynamic coupling measures and other class-level size and static coupling measures. However, dynamic coupling requires extensive test suites to exercise the system. Such test suites may not be readily available. We use the direct/indirect behavioral dependencies statically. In other words, we consider probabilistic and expected execution rate of messages based on the control structure and the operational profile without running a program.

6. Conclusion and Future Work

In this paper, we proposed the model-based behavioral dependency measurement between classes in UML 2.0 models of OO software for change-proneness prediction. We first provide definitions of the behavioral dependencies. Then, we suggest a systematic approach for calculating the BDM based on the defined behavioral dependencies. In the case study, the BDM has been shown to be an useful and complementary indicator compared to the model using only C&K metrics for change-proneness prediction. A model-based change-proneness prediction using BDM has several advantages, even the goodness of the model fit could be lower than code-based change-proneness prediction. At the early stage of the development, model-based change-proneness prediction provides a way of developing a stable and flexible software by helping the decision-making of design decisions or modifying current designs. It is much easier and cost-effective than reworking on the implemented system. Furthermore, it can be used in visualizing the spots of the changes, which would greatly improve the understandability of the software.

Some of our future works include: (1) extending BDM to take into account other dependency attributes such as time and impact. (2) investigating other applications of BDM such as fault-proneness prediction and object allocation in a distributed system. (3) visualizing change-prone classes on the modeling tool which is RSA. (4) confirming BDM's usefulness by applying it on the system which uses much of the inheritance and polymorphism.

ACKNOWLEDGEMENT

This research was supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Advancement)(IITA-2007-(C1090-0701-0032)) and this work was partially supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract.

References

[1] *Practiline Source Code Line Counter*. http://sourcecount.com/.

[2] *JFreeChart*, 2005. http://www.jfree.org/jfreechart.

[3] E. Arisholm, L. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transaction on Software Engineering*, 30:491–506, 2004.

[4] J. Bieman, A. Andrews, and H. Yang. Understanding change-proneness in OO software through visualization. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 44–53, 2003.

[5] S. Bohner and R. Arnold. Impact analysis - towards a framework for comparison. In *Proceedings of the International Conference on Software Maintenance*, pages 292–301, 1993.

[6] L. Briand, J.Wurst, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. *Proceedings of the International Conference on Software Maintenance*, pages 475–482, 1999.

[7] M. Chaumun, H. Kabaili, R. Keller, and F. Lustman. A change impact model for changeability assessment in object-oriented software systems. volume 45, pages 155–174. Elsevier, 2002.

[8] K. Chen and V. Rajich. RIPPLES: tool for change in legacy software. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 230–239, 2001.

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

[10] M. Gittens. *The Extended Operational Profile Model for Usage-based Software Testing*. Library and Archives Canada Bibliothèque et Archives Canada, 2005.

[11] L. Li, A. Offutt, and A. LLC. Algorithmic analysis of the impact of changes to object-orientedsoftware. In *Proceedings of the International Conference on Software Maintenance*, pages 171–184, 1996.

[12] N.Tsantalis, A. Chatzigeorgiou, and G. Stephanides. Predicting the probability of change in object-oriented systems. *IEEE Transaction on Software Engineering*, 31:601–614, 2005.

[13] D. L. Parnas. Some software engineering principles. pages 257–266, 2001.

[14] F. Xia. A change impact dependency measure for predicting the maintainability of source code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference*, volume 2, 2004.