World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:5, No:12, 2011

Quantifying the Stability of Software Systems via Simulation in Dependency Networks

Weifeng Pan, *Member, ACM*

Abstract—The stability of a software system is one of the most important quality attributes affecting the maintenance effort. Many techniques have been proposed to support the analysis of software stability at the architecture, file, and class level of software systems, but little effort has been made for that at the feature (i.e., method and attribute) level. And the assumptions the existing techniques based on always do not meet the practice to a certain degree. Considering that, in this paper, we present a novel metric, *Stability of Software* (*SoS*), to measure the stability of object-oriented software systems by software change propagation analysis using a simulation way in software dependency networks at feature level. The approach is evaluated by case studies on eight open source Java programs using different software structures (one employs design patterns versus one does not) for the same object-oriented program. The results of the case studies validate the effectiveness of the proposed metric. The approach has been fully automated by a tool written in Java.

Keywords—Software stability, change propagation, design pattern, software maintenance, object-oriented (OO) software.

I. Introduction

SOFTWARE maintenance is characterized as an activity of high cost, with typical estimates ranging from 60% to 80% of the total cost during the life cycle of the software systems [1]. The cost being so high, makes how to control the software maintenance cost an urgent as well as tough problem. In [2], Stephen S. Yau et al. proposed that there are generally two ways to control the cost. One is to provide some tools and techniques to help the maintenance practitioners perform their maintenance tasks. The other one is to utilize some meaningful software metrics. In this paper we will mainly focus on reducing maintenance cost through the utilization of metrics, i.e., to develop metrics to assess the quality characteristics of softwares affecting the software maintenance cost.

By the IEEE definition, software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [3]. And it has been regarded as a four-phase process in [2], [4]: (1) the first phase consists of analyzing a software system in order to understand it; (2) the seconde phase consists of generating a particular modification proposal to accomplish the implementation of the maintenance objective; (3) the third phase consists of accounting for the ripple effect as a consequence of software modifications; and (4) the fourth phase consists of testing the modified software to ensure the modified software has at least the same reliability as before. Therefore, we can find that performing software changes together with the change impact analysis (i.e., ripple effect analysis) correspondingly are two key activities in the software maintenance process, accounting for more than 40% of the total cost of software maintenance as reported in [5]. The primary attribute affecting the change impact analysis as a consequence of software modifications is the stability of the software [2], [4]. By stability of the software, it means the resistance to the amplification of changes in the software.

Software structure (i.e., topological structure) has a great influence on the quality of software systems [5]. In recent years, researchers in the field of statistical physics and complex system used complex software dependency networks to represent software systems by taking software components such as methods, classes and packages as nodes and their interactions as edges [6]. It provides us a new way to study complex software systems.

In this paper a novel metric, called *Stability of Software* (*SoS*), for quantitatively measuring the stability of Object-Oriented (OO) software systems using simulation in software dependency networks is presented. First, the software systems are modeled as weighted software dependency networks, weighted feature dependency networks (WFDN), in which features (i.e., methods and attributes) are nodes and the interaction between every pair of nodes if any is a directed edge which is annotated with a weight corresponding to the probability that a change in one features (method or attribute) will propagate to the other. Then we analyze the software change propagation process in WFDN via simulation, and based on which, *SoS* is developed to measure the stability of OO software systems. The results of the case studies on eight open source Java programs validate the effectiveness of the proposed metric. The approach has been fully automated by a tool written in Java.

The main contributions of this paper are summarized in the following:

(1) introduce WFDN to represent OO software systems at feature level and propose to use a probability way to perform the change propagation analysis;

(2) propose a catalog of changes with respect to WFDN to represent the changes in OO software systems;

(3) produce *SoS* based on change propagation analysis to characterize the stability of OO software systems;

(4) finally present a simulation approach to calculate *SoS* of OO software systems.

The rest of this paper is organized as follows. Section II contains a brief summary of the related work. Section III describes our approach in detail. Section IV presents the results of case studies conducted on eight Java programs. In

Weifeng Pan (Corresponding Author) is with the School of Computer Science and Information Engineering, Zhejiang Gongshang University, Hangzhou 310018, Zhejiang, P. R. China, e-mail: wfpan@mail.zjgsu.edu.cn.

Section V a software tool developed to automate the proposed approach is briefly described. The limitations and future work of our research work are mentioned in Section VI. And we conclude the paper in Section VII.

II. Related Work

This section is a brief, but, for reasons of space, incomplete, overview of the related work. It falls into two categories: (1) research work on software change impact analysis, and (2) research work on software dependency networks, both of which are the basis of the proposed stability metric.

A. Software Change Impact Analysis

Software change impact analysis estimates and determines the parts of a software system and related documentation if proposed software changes are made [7]. It is of vital importance to improve the overall efficiency in software maintenance. In [8], Kung et al. explored the change impact analysis in a class diagram. They introduced a novel notion, class firewall, to denote classes that may be impacted by a change in a given class. And the change impact analysis results have been used to address the test order problem in regression testing. In [9], Alan MacCormack et al. adopt Design Structure Matrices (DSMs) to represent the source files and their dependencies, and introduced the concept of change cost to measure the average influence of components on the whole system. Tsantalis et al. introduced a probabilistic approach to evaluate the potential flexibility of a given object-oriented software system [10]. This approach consider the types of changes and differs with the internal axis and external axis changes. Shaik et al. introduced change propagation coefficient to assess the design quality of software architecture [11]. In [12], we introduced an efficient statistical measure, called average propagation ratio, to analyze the change propagation process in the complex software networks at the granularity level of class. Li et al. proposed a software change propagation model and several metrics to characterize the change propagation process [13].

B. Software Dependency Networks

The quality of a software system is partially determined by its topological structure. So the need to reveal the internal relationships between structure and quality has become eminent. Recently, researchers used complex software dependency networks to represent software systems by taking software components such as methods, classes and packages as nodes and their interactions as edges. And many kinds of software dependency networks have been defined, such as software networks at package level [6], [14], software networks at class level [15], [16], [17], software networks at feature level [18], and software networks at method level [14]. In [19], Li et al. gave a detailed review of the research work in such a new field.

III. The Approach

In the previous researches as we talked in Section II, the authors always make the following three assumptions in their change impact analysis processes: (1) the change in one software component such as one file or class will definitely propagate to other components that have relationships to the changed node directly or indirectly; (2) in one change session there is only one initial changed file or class; and (3) all the existing change impact analysis are performed at file or class level. These three assumptions the existing techniques based on, however, may sometimes not meet the practice. The rationale is threefold:

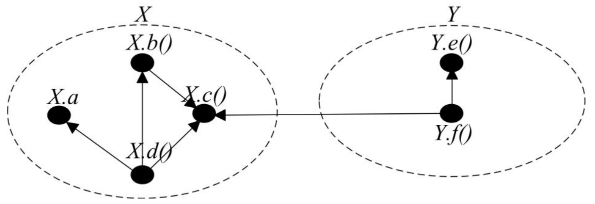


Fig. 1. A simple example

(1) In OO software systems, a class always contains many attributes and methods. We treat a class as changed if at least one of the methods or attributes in it changed. The attributes and methods of another class depending on the changed class do not all link to the changed attributes or methods directly or indirectly in it. So it does not always meet the practice that the change in one class definitely propagates to classes having relationships to it.

See figure 1, class X is composed of three methods (i.e., $b()$, $c()$ and $d()$) and one attribute (i.e., a) and class Y is composed of two methods (i.e., $e()$ and $f()$). So any attributes or methods of class X changes, class X will be viewed as changed. In previous work, authors all think the change in class X will definitely propagate to class Y, for the latter depends on the former. However it is not always the truth. For instance, if the change exists in method $c()$, the change may propagate to class Y, for $f()$ in class Y depends on $c()$ in X. However if the change exists in other attributes or methods but $c()$, it will not propagate to class Y, for $f()$ and $e()$ in class Y do not directly or indirectly depend on the changed attributes or methods in class X. So we should introduce probability in change impact analysis at class level. This situation also fits in with that at file level.

(2) Software systems will change from time to time in its life cycle. The change requirements will result in many parts of the system to be changed, i.e., the number of initial changed software components may be over one. So supposing there is only one changed software component in a change session does not meet the practice.

See figure 2, the nodes colored red, $X1$ and $X2$, are the initial changed classes in one change session. In the foregoing literatures, the classes in dashed rectangle will be counted twice. So it overestimates the results of change impact analysis.

(3) Complex OO software systems are generally composed of a number of classes which in turn contain many attributes and methods. And the change in one class will finally be transferred to changes in its attributes and methods. So we can do the change impact analysis at the feature level.

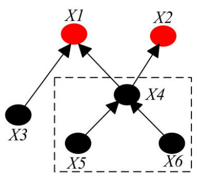


Fig. 2. A simple example

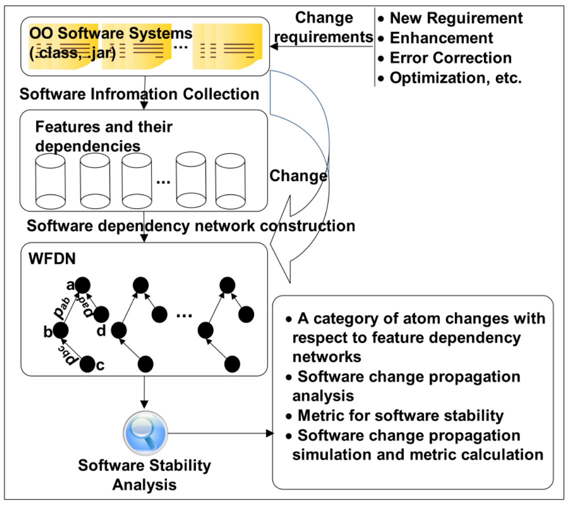

OO Software Systems (.class, .jar)
Change requirements
• New Reguirement
• Enhancement
• Error Correction
• Optimization, etc.
Software Infromation Collection
Features and their dependencies
Change
Software dependency network construction
WFDN
a
b
c
d
Software Stability Analysis
• A category of atom changes with respect to feature dependency networks
• Software change propagation analysis
• Metric for software stability
• Software change propagation simulation and metric calculation

Fig. 3. Workflow of the proposed approach

Our current work can be seen as an extension of the work done in [12], [13]. And in this paper we have primarily solved the demerits in existing research work mentioned above, i.e., we introduce probability when analyzing the change propagate from one class to others, take into consideration the situation that more than one initial changes occurred, and analyze the change propagation at the feature level. The overall framework of the proposed approach is illustrated in figure 3. The main steps in the framework will be specified as follows.

A. *OO Software Systems*

We mainly focus on the OO domains herein, and take the open source OO (hereafter OSOO) software systems as our research subjects. The rationale is twofold [18]: (1) OO has become the most widely used development paradigm since 1990's. And there are a lot of OSOO software systems on the web which can be easily got for our research objectives; and (2) OSOO software systems are developed under the OO paradigm. They have relatively clear internal structures and the components such as attributes, methods, classes, packages, and their dependencies are amenable to extraction and analysis.

B. *Software Information Collection*

Software information collection refers to the process to extract software components such as attributes, methods, and their dependencies. We have developed a tool that can be used to analyze compiled Java codes (.class and .jar) to get needed data. In this paper, we only take into consideration two kinds of dependencies, method accessing attribute dependency and method call dependency. Software information collection has been automated by a tool developed using Java (see Section V).

C. *Software Stability Analysis*

This subsection describes our approach in detail, with focus on the formal definition of the software dependency network, a list of atom changes with respect to software dependency network, the metric to characterize the stability of OO software systems, and the algorithm used to compute the proposed metric.

1) Software Dependency Network: After software information collection, the OO software systems can be modeled as one type of software dependency network, weighted feature dependency network. We use the term feature to designate attributes and methods. We will be treating them the same from here on. And the dependencies between two features as talked in subsection it B are treated as the same dependency, namely use dependency. We next give the formal definition of weighted feature dependency network.

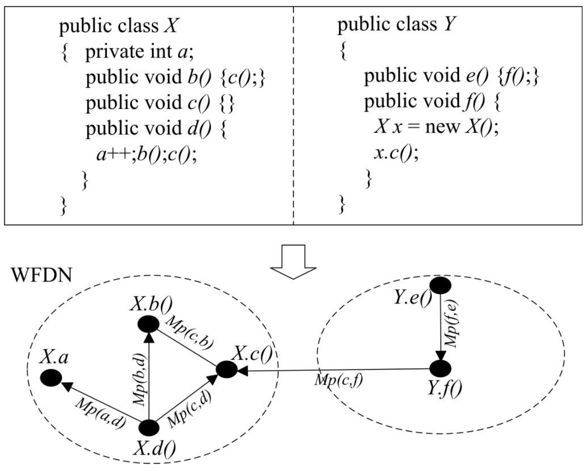
Definition 1: Weighted Feature Dependency Network

In Weighted Feature Dependency Network (WFDN), the nodes represent features (namely attributes or methods) of a specific OO software system. And each feature is represented by only one node. Edge between two nodes denotes one feature uses another feature. i.e., if feature A uses feature B, there will be an edge from the node denoting A to the node denoting B. And here we only consider the presence of dependency and neglect the multiplicity of dependencies such as A depends three times on B. And the weight of each edge denotes the probability that a change in B will propagate to A. See figure 4 for example. Since in our approach, the initial changed software components may be more than one, here we will also take into consideration the ratio that the initial changed nodes account for the total number of nodes in WFDN. Therefore WFDN can be described as:

$$WFDN = (N, E, M_p, CR), \tag{1}$$

where N is the set of all features of the specific OO software system; E is the set of edges denoting all relationships among features; M_p is a matrix storing the change propagation probability among all pairs of nodes if they are linked by an edge in WFDN, i.e., if node j links to node i, the entry $M_p(i,j)$ of M_p stores the probability that if node i changes, the change will propagate to node j with probability $M_p(i,j)$. If two nodes, node k and node l, have no edge linking them together, the entry $M_p(k,l)$ and $M_p(l,k)$ will be 0. CR is the ratio that the initial changed nodes (corresponding to the changed software components) account for the total nodes in WFDN. Figure 4 shows a simple source code segment and its corresponding WFDN.

In WFDN we assume the change probability between every pair of features that directly liked will be same, i.e., every no-zero entry in M_p will be same.



```
public class X
{  private int a;
   public void b() {c();}
   public void c() {}
   public void d() {
     a++;b();c();
   }
}
```

```
public class Y
{
   public void e() {f();}
   public void f() {
     X x = new X();
     x.c();
   }
}
```

Fig. 4. Illustration of WFDN

2) A Catalog of Changes: To use the software dependency network to analyze the software change impact, we should transform source code edits into a set of atomic changes. As talked in Section II, the authors also analyzed the change impact of OO software systems from the perspective of internal structure of softwares. They, however, fail to provide the list of atomic changes that can be applied in their method. We next, with reference to [20], present a catalog of atomic changes for OO software systems with respect to software dependency networks at feature level. We assume the original and the changed version of the OO software systems are both syntactically correct and compilable, and any source code change edits can be decomposed into a set of atomic changes as defined in table 1.

All these atomic changes in table 1 are self-explanatory. These atomic changes are presented in the perspective of WFDN and represent the source code edits at feature level. In our current work, we mainly analyze the stability of OO software systems by analysis on the existing software components in an OO software system. So atomic changes introducing new nodes and dependencies, like CMD and AF annotated"No", will be ignored in our current approach. In this point, our approach is no better than the existing work. And for the other atomic changes, regardless their nature, we will be treating them as the same change operation, namely change, and will be realized by random selection operations in our change propagation algorithm.

3) Change Propagation Algorithm: Any software system generally will undergo many times of modifications to adjust for the change requirements such as new requirements, enhancement, error correction and optimization. Since the diversity and randomness of software maintenance activities, it is of no meaning for the practitioners to predict when the next maintenance activity will occur and what this activity will consist of [2]. Because of the random nature of software maintenance activities, in our simulation, we will use the random selection operation to mimic this process. In the following subsections, an algorithm for software change impact analysis using simulation will be outlined. Before that some metrics used in this algorithm will be given first.

Definition 2: Change Probability Propagation Field of a Node, *CPPFN*

Suppose there is a specific WFDN. The $CPPFN$ of node i in this WFDN, $CPPFN(i)$, defines a set of nodes that are accurately affected by the change in node i in a specific simulation. And the size of this set can be denoted as $sCPPFN = |CPPFN(i)|$. Here and below, $|*|$ denotes the counting of the elements in set $*$.

Definition 3: Change Probability Propagation Field of a Set of Nodes, *CPPFSN*

Suppose there is a specific WFDN. The $CPPFSN$ of a set of nodes $setN$, $CPPFSN(setN)$, defines a set of nodes that may be affected by the change in nodes in $setN$ in a specific simulation. And the size of this set can be denoted as $sCPPFSN = |CPPFSN(setN)|$. Then the formula to calculate $CPPFSN(setN)$ is shown as:

$$CPPFSN(setN) = \bigcup_{\forall i \in setN} CPPFN(i) \qquad (2)$$

In **Algorithm 1** shown below, mp and cr are decimal numbers between 0 and 1, and $maxT$ is an integer far more than $|N|$; $simT$ is the number of simulation run times; $bChanged[]$ is an array with boolean type and each element of $bChanged[]$ stores the state of each node in WFDN, i.e., "true" denotes changed and "false" denotes unchanged; $bChangedBak[]$ is the backup array of $bChanged[]$; $bSelected[]$ is an array with type boolean and stores the state of each node, i.e., "true" denotes it has been selected as a node in initial changed node set and "false" denotes not; $CPPFSN[]$ is an array with an integer type and stores the $CPPFSN$ obtained in each simulation run; t is the counter of current run times; $cChgNN$ is the number of nodes that have selected as changed nodes in initial change set.

4) Metric for Software Stability: Based on the analysis above, here we will define a metric to characterize the stability of OO software systems.

Definition 4: Changed Node Ratio, *CNR*

Suppose there is a specific WFDN. CNR defines a ratio that the changed nodes account for the total nodes in WFDN from an initial state to a stable state in $simT$ simulations. And it can be calculated as:

$$CNR = \frac{\sum_{i=1}^{simT} CPPFSN[i]}{simT \times |N|} \times 100\% \qquad (3)$$

The notations have the same meaning as that used in change propagation algorithm.

Definition 5: Stability of Software, *SoS*

Then a novel metric for measuring the stability of OO software systems (hereafter SoS) can be produced, which can be computed according to formula (4):

$$SoS = 1 - CNR \qquad (4)$$

Obviously SoS is a scalar whose value between 0 and 1. A low SoS indicates a stable software where changes do not easily propagate between its software components.

TABLE I
CATEGORIES OF ATOMIC CHANGES

Abbreviation	Atomic Change Name	Realized?
DEM	Delete an empty method	Yes
DM	Delete a non-empty method	Yes
CHM	Change the header of a method	Yes
CM	Change the body of a method without new dependencies introduction	Yes
CMD	Change the body of a method with new dependencies introduction	No
AF	Add an attribute	No
DF	Delete an attribute	Yes
CF	Change an attribute without new dependencies introduction	Yes
CFD	Change an attribute with new dependencies introduction	Yes

Algorithm 1 Change Propagation Algorithm.

```
Input:
    WFDN, $mp$, $cr$, and $maxT$;
Output:
    $CPPFSN[i]$ ($i$ = 1, 2, ..., $|N|$);
1: Initial $M_p$, set each entry $M_p[i][j]$ = $mp$ if there is a direct edge from node $i$ to node $j$ in WFDN. Set $CR$ = $cr$ and $simT$ = $maxT$. Set $bChanged[i]$ = $false$, $bChangedBak[i]$ = $false$, $bSelected[i]$ = $false$, $CPPFSN[i]$ = 0 ($i$ = 1, 2, ..., $|N|$), and $t$ = 1. Set $cChgNN$ = 0. Prepare a queue cQueue.
2: If ($cChgNN$ $<=$ $cr \times |N|$) then randomly select a node $i$ which satisfies $bChangedBak[i]$ = $false$ and $bSelected[i]$ = $false$ from $N$. Push it into cQueue, set $bChanged[i]$ = $true$, $bChangedBak[i]$ = $true$, $bSelected[i]$ = $true$, $cChgNN$++, and go to step 3; else go to step 6.
3: If (cQueue is Null) then go to step 6; else go to step 4.
4: Pop one node from cQueue, denoted as $N_i$. Travel through the nodes in $N$ one by one (each node denoted as $N_j$), if ($Mp[i][j]$ = $mp$) then add node $j$ to a temporary set $tempSet$.
5: For each node $N_k$ in $tempSet$, randomly generate a decimal $dec_k$. If ($dec_k$ $>=$ $mp$), then (1) add $N_k$ into cQueue (i.e., the change in $N_i$ will propagate to $N_k$), (2) delete it from $tempSet$, (3) set the corresponding $bChanged[k]$ and $bChangedBak[k]$ of $N_k$ to be true, and (4) go to step 2; else delete it from $tempSet$ and go to step 2.
6: Set $CPPFSN[t]$ to be the number of non-zero elements of array $bChangedBak[l]$ ($l$ = 1, 2, ..., $|N|$). Set $t$ = $t$ + 1. If ($t$ < $simT$), then set $bChanged[i]$ = $false$ and $bChangedBak[i]$ = $false$ ($i$ = 1, 2, ..., $|N|$) and go to step 2; else go to step 7.
7: return $CPPFSN[i]$ ($i$ = 1, 2, ..., $|N|$).
```

5) Analysis on the Convergence of SoS: As talked above, SoS is computed using a simulation method. So, though the parameters like mp, cr, etc., are set to the same values in two separate runs, the SoS obtained may be different. So if we want to use SoS to characterize the stability of OO software systems, we should make a judgement that whether

TABLE II
JUNIT 3.4 STATISTICS

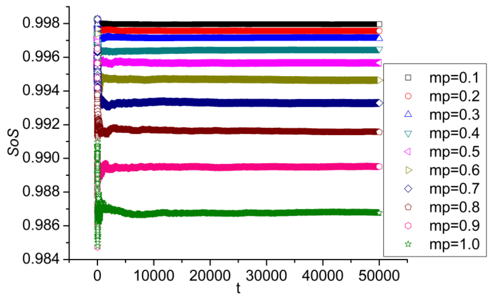
Parameter	Value	Parameter	Value
Number of packages	7	Number of classes	78
Number of features	601	$\|N\|$	575
$\|E\|$	889		

the proposed SoS can convergence to a relative stable value.

There are two main parameters (i.e., mp and cr) that should be set before running the change propagation algorithm. We next analyze the convergence of SoS by different settings of mp and cr with $maxT$ being same. And we use an OSOO software system, JUnit 3.4 [21] as our research subject. Table 2 shows the statistics of JUnit 3.4, including the number of packages, classes, and features of the whole systems. And here we focus on the WFDN composed of weakly connected components (WCC) with the number of nodes in each WCC larger than 1, i.e., the isolate nodes who have no direct edges to other nodes will be ignored. The number of nodes ($|N|$) and edges ($|E|$) in WFDN are also shown in table 2.

According to our experience in our daily work, the number of the initial changed nodes always will not be larger than 6. So here we set cr from 1/575 to 6/575 at interval 1/575 and $maxT$ = 50,000. And for each cr setting, we analyze the SoS vs. current run time t for mp ranging from 0.1 to 1 at interval 0.1. For limitation of space, here we only show the results of two simulations under two specific cr settings where cr = 1/575 and cr = 6/575. Please see figure 5 for illustration. From the curve of SoS vs. t, we can make the following four observations: (1) at the early period of simulation (especially t < 5,000), the SoS are not stable, fluctuating drastically; (2) when t is much more lager than $|N|$, like $t >=$ 50,000 in figure 5, SoS converges to a relative stable value; (3) under the same cr and t, the larger mp, the smaller stable SoS we obtained; and (4) under the same mp and t, the larger cr, the smaller stable SoS we obtained. In simulations with other mp and cr settings, we obtain the similar results.

But whether the observations obtained in JUnit 3.4 fit in well with that in other software systems? To answer this question, we randomly select about 100 software systems and analyze the stability of them using simulation method. And we also make the observations in all these software systems as that in JUnit 3.4. For limitation of space, here we omit the details



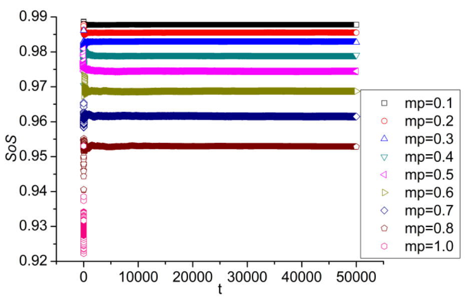


Fig. 5. Illustration of SoS vs. t. The left part: SoS vs. t with cr = 1/575 and $maxT$ = 50,000. The right part: SoS vs. t with cr = 6/575 and $maxT$ = 50,000

of the software systems used and the simulations on them. Based on the analysis above, we can conclude that SoS can be used as a metric to characterize the stability of OO software systems.

IV. Case Study

Design patterns are generally defined as descriptions of communicating classes that form a common solution to a type of design problem. They are widely accepted as a proven way to improve software quality [22]. Such an improvement in software quality should be qualitatively captured by the proposed metric, SoS.

A. Data Source

In order to investigate the applicability of the SoS proposed in this paper, eight Java programs have been examined. Both programs have two versions: one employs design patterns and one does not. These Java programs have been selected for analysis because they satisfy the following criteria: (1) they are written in Java which can be supported by our analysis tool; (2) the two versions of each Java programs have the same functionality, and the only difference is whether use design pattern or not; and (3) only one kind of design pattern will be used in one version of a software system. The design pattern used in one version of each Java program is Adapter, Bridge, Builder, Chain, Composite, Interpreter, Iterator, and Sate [23]. The source codes of the eight Java programs with two versions before and after using design pattern are available for download from [24]. Table 3 shows the statistics of the eight Java programs under study.

B. Results and Discussion

To analyze the stability of object-oriented software systems, we model them by WFDNs, using our own developed analysis tool SSAT (that will be detailed in Section V). To make it clear, for instance, figure 6 gives an illustration of the WFDNs of the Java program before and after using Adapter design pattern. Then, we apply our simulation method on the WFDNs of software systems under study and their SoS values are calculated. In all our simulations, $simT$ are all set to be 10,000. Table 4 shows the SoS of the eight Java programs

TABLE III
Statistics of the Eight Java Applications

	WFDN	WFDN	WFDN	WFDN
	Before	Before	After	After
Design Pattern	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$
Adapter	5	4	12	11
Bridge	26	48	41	65
Builder	11	13	31	34
Chain	7	8	10	13
Composite	11	15	12	15
Interpreter	4	3	20	25
Iterator	5	6	15	21
Sate	5	5	15	16

before and after using design pattern with $maxT$ = 50,000, $cr \times |N|$ = 1, and cr = 0.2α (α= 1, 2, 3, 4, 5). The results of simulations under other cr and mp settings all have similar conclusions. So for the limitation of space, here we omit them. For details, please refer to [24] where we have attached all the data used in this paper.

From Table 4, we can find that the SoS of Java programs using design patterns are larger than that of Java programs not using design patterns. The results matches with the anticipation that design patterns can improve the quality of software systems, and it verifies that the proposed method has the same ability of that in [10], [13].

V. Implementation

We have developed a Java program named Software Stability Analysis Tool (SSAT) adapted from SNAT in [18], which is mainly consists of three parts: (1) a bytecode parser, (2) a NET generator and parser, and (3) a SoS calculator.

The bytecode parser can parse the complied Java code (.class and .jar) to reveal the static structure WFDN, and store them in a file with NET file extension.

The NET generator, after the complied Java code has been parsed, produces a NET file, denoting WFDN. The NET file contains the information about the full feature names, the dependencies among them and the weight of each dependency edge (i.e., change propagation probability). It has the same format as that used in Pajek [25]. So you can also use Pajek to

TABLE IV
SoS OF THE EIGHT JAVA PROGRAMS WITH $maxT = 50,000$, $cr \times |N|$=1, AND mp=0.2α (α=1,2,3,4,5)

Design Pattern	mp=0.2		mp=0.4		mp=0.6		mp=0.8		mp=1.0	
	Before	After	Before	After	Before	After	Before	After	Before	After
Adapter	0.76726	0.900075	0.73698	0.881	0.7031	0.860717	0.67144	0.83775	0.63974	0.812883
Bridge	0.944435	0.967037	0.924104	0.95579	0.901631	0.943246	0.879369	0.929088	0.858127	0.913768
Builder	0.886064	0.960255	0.860818	0.953065	0.829336	0.944642	0.801873	0.936771	0.776091	0.927984
Chain	0.821386	0.87126	0.778914	0.8382	0.733714	0.80083	0.683729	0.76639	0.631786	0.7293
Composite	0.882482	0.892342	0.847091	0.866942	0.808427	0.837133	0.770873	0.807825	0.734336	0.777725
Interpreter	0.7102	0.936635	0.663675	0.9216	0.613675	0.902025	0.5598	0.8816	0.49895	0.85966
Iterator	0.7471	0.911807	0.68474	0.88482	0.628	0.853267	0.56862	0.822547	0.51544	0.792087
Sate	0.75778	0.91716	0.70876	0.90002	0.65586	0.878807	0.60412	0.8554	0.5623	0.83218

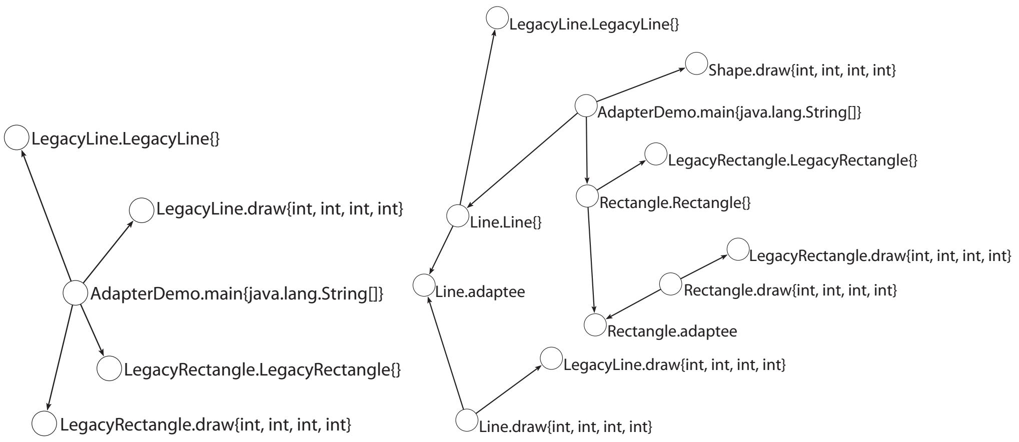
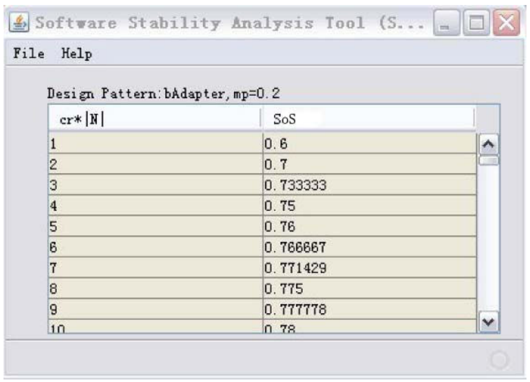

LegacyLine.LegacyLine{}
LegacyLine.draw{int, int, int, int}
AdapterDemo.main{java.lang.String[]}
LegacyRectangle.LegacyRectangle{}
LegacyRectangle.draw{int, int, int, int}
LegacyLine.LegacyLine{}
Shape.draw{int, int, int, int}
AdapterDemo.main{java.lang.String[]}
LegacyRectangle.LegacyRectangle{}
Rectangle.Rectangle{}
Line.Line{}
LegacyRectangle.draw{int, int, int, int}
Line.adaptee
Rectangle.draw{int, int, int, int}
Rectangle.adaptee
LegacyLine.draw{int, int, int, int}
Line.draw{int, int, int, int}

Fig. 6. Illustration of WFDNs of the Java program before (left) and after (right) using adapter design pattern.



Design Pattern:bAdapter,mp=0.2

cr*|N|	SoS
1	0.6
2	0.7
3	0.733333
4	0.75
5	0.76
6	0.766667
7	0.771429
8	0.775
9	0.777778
10	0.78

Fig. 7. Sample screenshot for SSAT

give an illustration of the WFDN. For the limitation of space, here we will not go further into details about the format of the NET file.

The SoS calculator applies the aforementioned approach to calculate SoS for a specific software system.

A sample screenshot of SSAT applied to the Java program before using adapter design pattern with $maxT$ = 50,000, $cr \times |N|$ = 1, and mp = 0.2, is shown in figure 7.

VI. LIMITATIONS AND FUTURE WORK

Although our approach shows some feasibilities in measuring the stability of the sample Java programs, the broad validity of our approach demands further demonstration. Moreover, when constructing WFDN, we suppose that the change in one feature will propagate to other features with the same probability. This may not meet the practice in some circumstance.

Thus, the future work includes:

(1) validating the approach using more other open source software systems written in Java and other programming languages (e.g., C++, C#);

(2) presenting a more realistic approach which takes into considering in WFDN the non-trivial probability (not simply the same probability).

VII. CONCLUSION

In this paper we used the weighted feature dependency network (WFDN) to model the topological structure of OO software systems, examined the change propagation process in WFDN using a simulation way, and finally proposed a metric SoS to characterize the stability of OO software systems. The rationale behind this approach is that in a high quality software system, changes arising in features should be limited to a range as small as possible, i.e., SoS should be kept as small as possible.

Case studies have shown the effectiveness of SoS in software stability measurement. The proposed approach improves the accuracy of existing methodologies. And it has been automated by a tool written in Java and can be applied to measure the SoS of any OO software system written in Java.

ACKNOWLEDGMENT

This research is supported by the National Basic Research 973 Program of China under Grant No. 2007CB310801, the National Natural Science Foundation of China under Grant Nos. 60873083, 60903034, 61003073 and 61070013, the Natural Science foundation of Hubei Province under Grant Nos. 2008CDB351 and 2008ABA379, the Research Fund for the Doctoral Program of Higher Education of China under Grant No. 20090141120022, and the Fundamental Research Funds for the Central Universities of China under Grant No. 6082005.

REFERENCES

[1] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*. New York, England: McGraw-Hill, 1992.
[2] S. Yau and J. S. Collofello, "Design stability measures for software maintenance," *IEEE Transactions on Software Engineering*, vol. 11, no. 9, pp. 849-856, 1985.
[3] IEEE Std. 610.12, *Standard Glossary of Software Engineering Terminology*, IEEE Computer Socienty Press, Los Alamitos, CA, 1990.
[4] S. Yau and J. S. Collofello, "Some Stability Measures for Software Maintenance," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 545-552, 1980.
[5] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2rd ed. Londom, UK: International Thomson Computer Press, 1996.
[6] C R. Myers, "Software systems as complex networks: Structure function, and evolvability of software collabration graphs", *Physical Review E*, 2003, 68: 046116.
[7] S. Bohner and R. Arnold, *Software Change Impact Analysis*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
[8] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima and C. Chen, "Change Impact Identification in Object-Oriented Software Maintenance," *Proc. of IEEE International Conference on Software Maintenance*, pp. 202-211, 1994.
[9] A. MacCormack, J. Rusnak and C. Y. Bald Win, "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary code," *Management Science*, vol. 52, no. 7, pp. 1015-1030, 2006.
[10] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides and I. Delegiannis, "Probabilistic Evaluation of Object-Oriented System," *Pro. of the 10th International Symposium on Software metrics*, pp. 26-33, 2004.
[11] I. Shaik, W. Abdelmoez, R. Gunnalan, A. Mili, and C. Fuhrman, "Using change propagation probabilities to assess quality attributes of software architectures," *Proc. of IEEE International Conference on Computer Systems and Applications*, pp. 704-711, March, 2006.
[12] J. Liu, J. Lu, K. He, B. Li and C. K. TSE, "Characterizing the structural quality of general complex software networks via statistical propagation dynamics," *International Journal of Bifurcation and Chaos*, vol. 18, no. 4, 2008.
[13] L. Li, G. Qian and L. Zhang, "Evaluation of software change propagation using simulation," *Proc. of World Coongress on Software Engineering 2009*, pp. 28-33, May 19-21, 2009.
[14] D. Hyland-Wood, D. Carrington and S. Kaplan, "Scale-free nature of Java software package, class and method collaboration graphs," *Technical Report of MiND Laboratory*, 2006, No. TR-MS1286, University of Maryland College Park, 2006.
[15] H. Li, B. Huang and J. Lu, "Dynamical evolution analysis of the object-oriented software systems," *Proc. of 2008 IEEE Congress on Evolutionary Computation*, pp. 3030-3035, June 1-6, 2008.
[16] H. Li, "Scale-free networks models with accelerating growth," *Frontiers of Computer Science in China*, vol. 3, no. 3, pp. 3030-3035, 2009.
[17] J. Liu, K. He, Y. Ma and R. Peng, "Scale free in software metrics," *Proc. of IEEE Proceedings of 30th Annual International Computer Software and Applications Conference*, pp. 229-235, Sept. 18-21, 2006.
[18] W. F. Pan, B. Li, Y. T. Ma, J. Liu and Y. Y. Qin, "Class structure refactoring of object-oriented softwares using community detection in dependency networks," *Frontiers of Computer Science in China*, vol. 3, no. 3, pp. 396-404, 2009.
[19] B. Li, Y. T. Ma and J. Liu, "Advances in the studies on complex networks of software systems," *Advances in Mechanics*, vol. 38, no. 6, pp. 805-814, 2008.
[20] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," *Proc. of ACM SIGPLAN-SIGSOFT Workshop on Program analysis for software tools and engineering*, pp. 45-53, 2001.
[21] JUnit, http://junit.sourceforge.net, May 5, 2010.
[22] N. Tsantalis, E. Chatzigeorgous, G. Stephanides and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896-909, Nov., 2006.
[23] R. Martin, Design principles and design pattern. http://www.objectmentor.com, May 5, 2010.
[24] Data for the case studies, http://blog.sina.com.cn/breezepan, May 5, 2010.
[25] Pajek, http://pajek.imfm.si/doku.php, May 5, 2010.

Weifeng Pan received his Ph.D. degree from State Key Laboratory of Software Engineering at Wuhan University in 2011. His current research interests include software engineering, service computing, complex networks, and intelligent computation. He is a member of China Computer Federation (CCF), and Association for Computing Machinery (ACM). In June 2011, he joined School of Computer Science and Information Engineering, Zhejiang Gongshang University.