International Journal of Computer Applications (0975 – 8887)
Volume 36– No.6, December 2011

# Identifying the Dissimilarities based on Working of Programs among Versions in DVCS (Distributed Version Control Systems)

Laika Satish
Faculty of Computing and Information technology,
King Abdul Aziz University,
Rabigh, Saudi Arabia

## ABSTRACT

In this paper, one of the most important phases of Software development that is versioning, which is done through version control systems, is being presented. The current methodologies used in distributed version management, some aspects needed in the working of Version Control Systems, the prior work that is done in this field of technology are discussed. A proposed algorithmic approach for knowing the dependency of linkages of classes, interfaces and methods in object-oriented technology amid versions of a program is also presented. The aim of this approach is to decrease errors and inaccuracies during the phase of software development in distributed version control systems.

## General Terms

Software engineering, Software maintenance, Software configuration Management.

## Keywords

Tracking repository changes, Differences based on working aspects of programs.

## 1. INTRODUCTION

Software development and maintenance is a process that is performed by a group of developers. Regarding cost, the requirement for tools and methods to aid in the software development arouses naturally [3]. The high cost of software maintenance makes it important to look into the inefficiencies and make new methods to improve the effectiveness of the maintenance process. This requires proper look through into the entire application, which are stored in repositories called version control systems.[2]

Version Control System (VCS) is a centralized place, which stores source code of the application part of the software system. A VCS helps to synchronize the work of many developers working at the same time on the product. The VCS also maintains a complete history of changes in the code of the software system. Every change is represented as a special record in the VCS called a check-in. For every check-in, the VCS stores the date when each change is made, the developer name who made the code change, a description of the change, the list of changed source files, and the actual changes in source code in the form of a textual diff. To every check in, a check-in ID is allocated. Each check-in changes one or more source files. To differentiate between different versions of these files, version numbers are allotted.

Thus the VCS, is very important source of information in the project lifecycle. They control data on source files, functions, check-ins, bug records, and entire team of developers. Distributed revision control (DRCS) follows a peer-to-peer approach, as opposed to the client-server approach of centralized systems[17] .For example, VCS ensures that the name of the developer is always there in the check-in.

Thus through VCS, most of the problems that are encountered are solved but working out of these changes are limited in object-oriented programming. This technique helps us to identify object-oriented features and, works on comparing changes based on functionality of object-oriented programs.

Distributed computing means, where programs are split among a grid of machines. Distributed version control gives importance to on sharing changes; every change has a unique id[1].

## 2. THE MAINTENANCE PHASE OF SOFTWARE DEVELOPMENT

The maintenance phase mostly involves change management also referred to as software configuration management (SCM).

It is like an umbrella activity that is done during the software process. Its aim is to increase productivity by decreasing inaccuracy and errors during software development [18].

New market demands require changes in software requirements or new business rules may come into focus demanding changes in functionalities in software.[10]

Even if software tools have the potential to progress in maintaining the quality of software, in a large organization this can be an expensive scheme Thus**,** Even cost or time constraints may need a redefinition of the product. In Centralised systems, it is a central repository where developers can check in and check out**,** but the centralised systems does not achieve the required mark for merging and branching changes .In distributed systems, object oriented programs may be split into multiple portions for example an interface may be on server 1 whereas the class which implements that interface may be on server2.distributed systems make branching and merging more manageable in comparison to centralized version control systems **.**

Software engineering involves analysis of multiple versions of a program. The development team needs to Store data in form of a tree, which can be easily understood, have access to the entire source code. Developers should be able to work in synchronized manner at the same time on the same files (Users edit their working copy locally), keep track of different versions of the same file (history)

Thus, designed for concurrent editing and to store history information. Thereby granting read/write access to the stored data. For all these purposes a repository is used which is like a centralized place to store data:[4]

Through a repository, a programmer can see a version of the program at any time in the history of the project and know how those files looked like at the time. He can also see the latest version of the file from the repository.

## 3. THE CURRENT METHODOLOGY OF VERSION MANAGEMENT

In the process of software engineering, information about changes between different versions of a program is needed in each step of the software lifecycle. For these there are many algorithms currently. Differencing algorithms give information about the places of the program where changes have being made [11].

Changes anywhere in the application have an impact on software engineering tasks. For example, whenever there is a code modification. Testing needs to be redone, which effects the estimation of time and cost of the software programs. Huge projects on which many developers are working need a Version Control System. There are many tools available that give us the changes between n number of files like diff, SCCS (Source Code Control system for unix), RCS (RCS also for unix), CVS (Concurrent version system for windows). In distributed we have systems like GIT, INFOQ, Mercurial, Bazaar.

A Version Control System does the following:

**Long-term undo.**

Suppose we made a change a year ago, and it had a bug we want to go back to the old version, and check what changes were made on that day.

**Branching and merging**.

During check in we can **merge** our work back into the common area.

**Working Set/Working Copy**: the local directory of files, where we make changes.

**Trunk/Main**: The primary location for code in the repository. The code is like a family tree

— The trunk is the main line.

Actions in VCS

**Add**: Put a file into the version control system for the first time,

i.e. start using it with Version Control.

**Revision**: What version a file is on (v1, v2, v3, etc.).

**Head**: The latest revision in the VCS.

**Check out**: Download a file from the vcs.

**Check in**: Upload a file to the VCS.

The file gets a new revision number, and developers can “check out” the latest one.

**Checkin Message**: A short message describing what was changed.

**Change log/History**: A list of changes made to a file since it is created.

**Update/Sync**: Synchronize your files with the latest from the VCS.

**Revert**: Throw away the local changes and reload the latest version from the repository.

Advanced Actions

**Diff/Change/Delta**: Finding the differences between two files.

**Merge (or patch)**: Apply the changes from one file to another, to bring it up-to-date.

**Resolve**: Resolving the changes contradicting each other and

checking in the correct version.

**Locking**: giving access control to a file so nobody else can edit it unless it is unlocked.

**Check out for edit**: Checking out an “editable” version of a file [13].

## 4. THE NEED OF VERSION MANAGEMENT BASED ON FUNCTIONALITY

The aim of the tools used in version management is to collect information about the application spread in these different sources and give the functionality needed by the developer in its current state or intermediate states that the developer requires in each step of software engineering, just relying on syntactic differences in object oriented programs does not go well for software developers. [14]

In this paper, I present a technique for comparing programs in objected oriented disciplines in distributed architecture that identifies differences based on functionality between n versions of the program.

Thus, I present an algorithm that acquire information from a version control system and track changes based on functionality This algorithm allows retrieving information on code integrations not only for recent code changes, but for huge amounts of versioned data too.

Given two lines of code, A and B, i.e. if coupling exists B must change behavior only when A is changed.

"Functional changes" refer to source code changes, or run-time results? It refers to explicit changes in working - i.e. the source code. [17]

**These functional program changes are limited to OO programming.**

When we manage changes instead of managing versions, merging works better, and therefore, you can branch any time your organizational goals require it, because merging back will be a piece of cake. [19]

```
interface I1 {
  abstract void test(int i);
}
interface I2 {
  abstract void test(String s);
}

public class Example1 implements I1, I2
{
  public void test(int i) {
    System.out.println(i);
  }
  public void test(String s) {
    System.out.println(s);
  }

  public static void main(String[] a) {
    Example1 t = new Example1 ();
    t.test(3);
    t.test("Course code COCS 202");
  }
}
```

Fig.1

```
interface I1 {
            abstract void test1(int i,int j);

}
interface I2 {
             abstract void test(String s);
}
class Example2
{
   void show(int a,int b)
   {
            System.out.println("Lecture and lab credits
            units are "+a +" "+b);
   }
}

public class Example1 extends Example2 implements I1, I2
{
   void show(int a,int b)
   {
      int TotalCredits=a+b;
       System.out.println("TotalCredits are     "+TotalCredits);
   }
 public void test1(int i,int j) {
   System.out.println(i);
   System.out.println(j);
 }
 public void test(String s) {
   System.out.println(s);
 }
  }
 public static void main(String[] a) {
   Example1 t = new Example1 ();
   t.test1(3,2);
   t.test("Course code COCS 202,COCS 203");

}
}
```

## 5. THE PROPOSED ALGORITHMIC APPROACH FOR KNOWING THE DISSIMILARITIES BETWEEN VERSIONS BASED ON FUNCTIONALITY

I present a small example here as shown in fig 1. implemented in ASP.net.There are 2 versions of the same file .in file 1, There are 2 interfaces ‘I1’ and ‘I2’ and a class ‘Example1’ which has 2 methods from the interface and the implementation of these methods.

In version 2, we have the same 2 interfaces I1 and I2 with the deletion of method test in interface I1 and addition of a new method test1, which takes 2 parameters of int datatype. We also have a new class called Example 2 which also has a method named show and the implementation of the method .We have the same class Example 1 which now extends Example 2 and implements the interface methods and also has methods from the parent class Example 2.

Consider in case of huge applications, new developers may replace old ones .If they want to know the functionality between these versions other than just semantic differences, they need something more in version control systems.

So, I present the following method, which will take

```
Input: checked in versions $V_1$, V2…VN of the application in the development trunk in the servers $s_1$ ….$s_n$
Return:  dependency of classes (parent and child classes), methods, interfaces in each version in the development trunk in the server’s $s_1$ ….$s_n$
Begin
For each checked in version $V_{1...}$ $V_N$ in the trunk
        Sort v [] in the ascending order and add it in a stack.
End
For each file f in v []
Compare classes in each file f
Locate newly added and removed classes (parent classes and child classes), interfaces, and methods in each version.
Check for methods common in each class in the versions /*this will give us information about the changes in each method */

Check for semantic differences in each class using the diff tool.

End

 Construct a graph in which each vertex may correspond to a parent class or a child class or an interface or a method and each edge will correspond to flow of dependence between each other or with the main class existing on server’s $s_1$ ….$s_n$.

Mark each edge in the proper order with an ordering number.

For every edge traversed, make a string format, which consists of the ordering number on the start point, ordering number on the end point, edge value, and serverId.

Hash the value coming from the above step.
Return the set of hash values  /* which help us to know the dependency of each class and methods in these classes. */
```

## 6. COMPARATIVE STUDY OF EXISTING TECHNIQUES AND EXPERIMENTAL RESULTS OF THIS APPROACH

Apiwattanapong et al. came up with the CalcDiff algorithm, which works, on the behaviors of object-oriented programs. For this they used ECFG’s (Enhanced Control flow graphs) in centralized version control systems but it doesnot show the differences based on working of programs. Thomas Horwitz and Susan Reps came up with an algorithm, which would make a program into slices evaluate these two program slices. This algorithm works on dependence graph of slices of the program. We also have many semantic differencing algorithms like cdiff and jdiff. Some version control systems like gits donot work good for windows. Mercurial also is complicated for beginners.

Bazaar is the topmost distributed version control systems but lacks this approach. So this approach gives the method linkages and changes in detail rather than just hash numbers and would be easy for a beginner to understand. This approach can be used for very huge projects.

## 7. EXPERIMENTAL RESULTS

I made a file comparison between 2 revisions and tested my implementation .The outcome gives us the line numbers of the 2nd revision about where all changes are made and also shows us the textual changes among both the revisions .It gives us the total number of lines also for each, class Example1, Example2 interface I1, I2 in both the revisions .For example, if class Example2, Interface I1, Interface I2 all exists on a different server it shows that Example 1 on server # extends Example 2 on server # implements interface I1 on server # and I2 on server#. The method differences are shown in both the revisions based on the set of hash values returned by the algorithm. Described below is the 2nd revision.

| **void show(int a, int b) exists in class Example 2  on server #** |
|---|
| Method signature exists in interface I1 on server#=False |
| Method signature exists in interface I2 on server#=False |
| Method extended from class Example 2 on server#=True |

| **void test1(int i,int j) exists in class Example 1 on server #** |
|---|
| Method signature exists in interface I1 on server#=True |
| Method signature exists in interface I2 on server#=False |
| Method extended from class Example 2 on server#=False |

| **void test(String s) exists in class Example 1 on server #** |
|---|
| Method signature exists in interface I1 on server#=False |
| Method signature exists in interface I2 on server#=True |
| Method extended from class Example 2 on server#=False. |

All the method hash numbers are not the same in both the revisions. It also gives us the number of methods in revision 1 and 2.

Total number of methods in revision1=#

Total number of methods in revision2=#

## 8. CONCLUSIONS

In this approach, a way is being illustrated for knowing the dependency of classes on interfaces or parent classes.

We represent the changed information in a way to help programmers deal with changes in the applications in the version control system. This approach will also help the programmers to write changes in a structured way. Handling comparisons only at the semantic level is not sufficient for maintaining very large projects.

## 9. REFERENCES

[1] Ian Clatworthy, Distributed version control systems why and how, Canonical.

[2] Sparx Systems, 2010,Version Control Best practices for enterprise architect.

[3] Prof.Stafford, Department of computer Science, Tufts University, Software Maintenance as part of the software lifecycle, software Engineering.

[4] Introduction to Distributed version control Better Explained, 2007.

[5] Takafumi ODA†*, Nonmember and Motoshi SAEKI†a), Member, 2006,Meta Modeling based version control system for software diagrams, IEICE TRANS. INF. & SYST.

[6] Wuu Yang, Susan Horwitz and Thomas Reps, 1989,a new program Integration algorithm, Computer Sciences Technical report.

[7] Hung-Fu Chang, Audris Mockus, University of Southern California. University Park Campus, Avaya Labs Research, Constructing Universal version History.

[8] James J Hunt, Kiem-Phong Vo and Walter F Tichy, University of Karlsruhe, Germany, An empirical study of delta algorithms.

[9] Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato, Version Control with Subversion

[10] Alexander Tarvo, Brown University, Thomas Zimmerman, Jacek Czerwonka, An integration resolution algorithm for Mining multiple branches in version control systems.

[11] Taweesup Apiwattanapong, Alessandro Orso, Mary Jean Harold differencing technique and tool for object oriented programs.

[12] David L. Atkins, Thomas Ball, Todd L. Graves_ and Audris Mockus, Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor

[13] Carlos Garcia Campos IT, A distributed version controls System.

[14] Miguel A. Figueroa Villanueva, Xabriel J. Collazo Mojica, Version Control Systems subversion, University of Puerto Rico.

[15] Alexander Yip, Benjie Chen and Robert Morris, MIT Computer Science and AI Laboratory, Past Watch a distributed version control system.

[16] Ted Naleid, Distributed version control system with mercurial.

[17] Version control systems, Wikipedia.

[18] Pressman, Software engineering, A practitioners approach.

[19] Distribution control by Joel spolsky.