An Integration Resolution Algorithm for Mining Multiple Branches in Version Control Systems

Alexander Tarvo*
Brown University
Providence, RI, USA
alexta@cs.brown.edu

Thomas Zimmermann, Jacek Czerwonka
Microsoft Corporation
Redmond, WA, USA
{tzimmer, jacekcz}@microsoft.com

Abstract.
The high cost of software maintenance necessitates methods to improve the efficiency of the maintenance process. Such methods typically need a vast amount of knowledge about a system, which is often mined from software repositories. Collecting this data becomes a challenge if the system was developed using multiple code branches.

In this paper, we present an *integration resolution algorithm* that facilitates data collection across multiple code branches. The algorithm tracks code integrations across different branches and associates code changes in the main development branch with corresponding changes in other branches. We provide evidence for the practical relevance of this algorithm during the development of the Windows Vista Service Pack 2.

1. Introduction

Maintenance of a large software system is a costly activity. It is estimated that maintenance expenses can account for the 50% of overall expenses associated with the software system lifecycle [1]. Given the tremendous cost of software maintenance, a large effort is undertaken to streamline software maintenance and make it less expensive. Popular approaches to improve efficiency are change impact analysis [2][3], models to predict fault-proneness [4][5], or models to predict software regressions [6][7]. These tools help improving efficiency of the software maintenance by directing testing efforts.

However, all these tools require a large amount of information on the software system. This information is collected with special tools called mining tools from various software repositories, such as version control systems, bug databases, and the program source code itself. The goal of mining tools is to collect information on the system scattered in these sources and restore a holistic picture of software system's evolution and its current state. Several mining tools have been developed [10][15], but they have one significant drawback: they do not properly handle the situation, when software systems use multiple code branches.

A *code branch* is a separate copy of the program's source code that evolves separately from the original copy [8]. In general, branches support parallel software development. Using branches is beneficial during both development and maintenance of the software system. On the one hand, code branches allow to efficiently distributing development across multiple teams. On the other hand, code branches facilitate efficient maintenance of the software system and help minimizing risk of undesired changes in the behavior of the software system. For example they can alleviate negative effects of software regressions, i.e., bugs which make features stop functioning after a change. However, branching poses some unique challenges when mining information on the evolution of a software system: code changes can migrate between different branches; this process is called *code integration*. As a result, it is not be possible to restore a complete picture of the change history by just looking at changes in only one branch. Unfortunately, many version control systems provide only limited information on code integrations. Thus analyzing multiple branches is a non-trivial task as we will show.

* Alexander Tarvo was an employee of Microsoft Corporation when this work was carried out.

In this paper we discuss problems associated with collecting data from multiple code branches used during development or maintenance of a system. To tackle these problems, we propose an integration resolution algorithm that facilitates collecting change data from multiple code branches and discuss some of its applications, such as analyzing periodic builds of the software system. To ensure high flexibility the integration resolution algorithm is implemented not as a part of the version control system, but rather as a stand-alone tool. On the one hand, this allows using the algorithm in a case when modification of the version control system is not possible, e.g. because its source code is not publicly available. On the other hand, it allows retrieving information on code integrations not only for recent code changes, but also for vast amounts of historical data.

Finally, we present our experiences of using this algorithm during the Windows Vista Service Pack 2 development. We also propose a set software metrics based on code integrations, which can help monitoring the progress of software development.

2. Related Work

Mining software repositories to collect information on evolution of the software system gained lots of attention recently. For example, Silwerski et al. [9] and Kim et al. [16] introduced the concept of bug-introducing changes. By mining code changes and fixes, they built a classifier that allows detecting buggy changes [17]. Cubranic and Murphy developed the Hipikat mining tool [10] that scans version control systems, bug-tracking database, and mailing list to extract information on entities, such as code changes, developers, and bug records. Hipikat identifies relations between these entities (e.g. what bug is responsible for a particular code change) and stores this information in relational form. The resulting data is used to educate and guide developers who are new to a project.

Mining information about code changes in multi-branch software projects has also drawn attention. Perry et al. [11] studied parallel and potentially conflicting changes, e.g. when many developers are editing the same source file. They proposed a number of metrics to quantify such a parallel development and showed that parallel development has a generally negative effect on the quality of the software. Zimmermann studied the frequency of code integrations in CVS repositories [19] and Bird et al. described merging practices for GIT repositories [18]. In an exploratory study on ArgoUML, Williams and Spacco [21] identified characteristics of code integrations.

Zimmermann and Weißgerber [12] describe how to effectively preprocess data for mining software repositories. They notice that code integrations from different branches can be considered as noise and call for an algorithm for detecting code integrations. One such algorithm is proposed in the work of Fischer et al. [13]. They point out that much of information about software evolution is lost during code integrations. Their heuristic-based algorithm can detect code integrations in the CVS version control system. However, it has certain assumptions, for example that code integrations occur only at the end of branch history. Mens [22] and Kim and Notkin [20] survey the research on software merging and differencing. However, most of this work addresses how to effectively compare and merge changes and not how to deal with integrations for software analysis and mining.

To the best of our knowledge no research has explicitly accounted for code integrations when mining software repositories. In most cases, code integrations are either ignored or simply not differentiated at all [23]. In this paper we show how to include information on code integrations for the analysis of software evolution. We propose an algorithm that reports not only the point of code integration, but also ***reports all the changes in different branches that were brought by that integration to the trunk branch***. In our work, we use information about integrations provided by the version control system itself. If such data is not available, one can combine our algorithm with the integration detection heuristic introduced by Fischer et al. [13]. Furthermore, we consider ***multi-branch scenarios***, i.e., when code changes flow through multiple branches before actually ending in the trunk.

3. Sources of information on the evolution of a software system

Development and maintenance of large software systems is a resource-consuming process that is usually performed by a large group of developers. Considering its cost and complexity, the need for tools and methods to aid in the software development arouses naturally.

One such tool is the *Version Control System* (*VCS*), a centralized database that stores source code of the software system. A VCS allows synchronizing the work of many developers working on the product and in addition to that, it also stores a complete history of changes in the source 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 of the change, the name of the developer who made the change, a short description of the change written in plain text, the list of changed source files, and the actual changes in source code in the form of a textual diff. Check-ins are identified by a unique number or a tag called a check-in ID. Each check-in changes one or more source files. To discriminate between different versions of these files, *version numbers* are used by the VCS. Such a version number, along with the full path to the source file, allows identifying every particular version of any source file in the system.

Check-ins provide details on the changes themselves, but they give only very limited information about the rationale why these changes were made. This information is typically stored in the *Bug-Tracking Database* (*BTD*) in the form of *bug records*. Bug records are used to track the work on various changes made in the code of the project, such as bug fixes, new features, maintenance or reliability improvements, and other changes. Every bug record within the BTD is identified by a unique number (called a bug ID) and contains useful information on the issue itself, including nature of the issue (is it a bug or a new feature), names of people who worked on the issue, security effect of the issue, and other data. Moreover, bug records contain information about software regressions. If the change for a bug report fixes a regression caused by some earlier change, the ID of the bug report that caused the regression is also recorded.

The VCS, BTD, and the code of the system are very important sources of information about the evolution of a project. They contain data on a variety of entities, such as binary modules, source files, functions, check-ins, bug records, and people. These entities are related to each other in many ways. For example, the bug is related to check-in if fixing the bug results in creation of the check-in. Similarly, the source file is related to the binary if that source file is used to build that binary (see Figure 1), and the check-in is related to the developer if the developer is one who created that check-in.

Some relations, such as check-in to developer relation, are straightforward to discover since VCS ensures that the name of the developer is always present in the check-in. Discovering other relations, e.g. bug to check-in relation, is not as easy. Although the ID of the corresponding bug is usually entered into the check-in description field by the

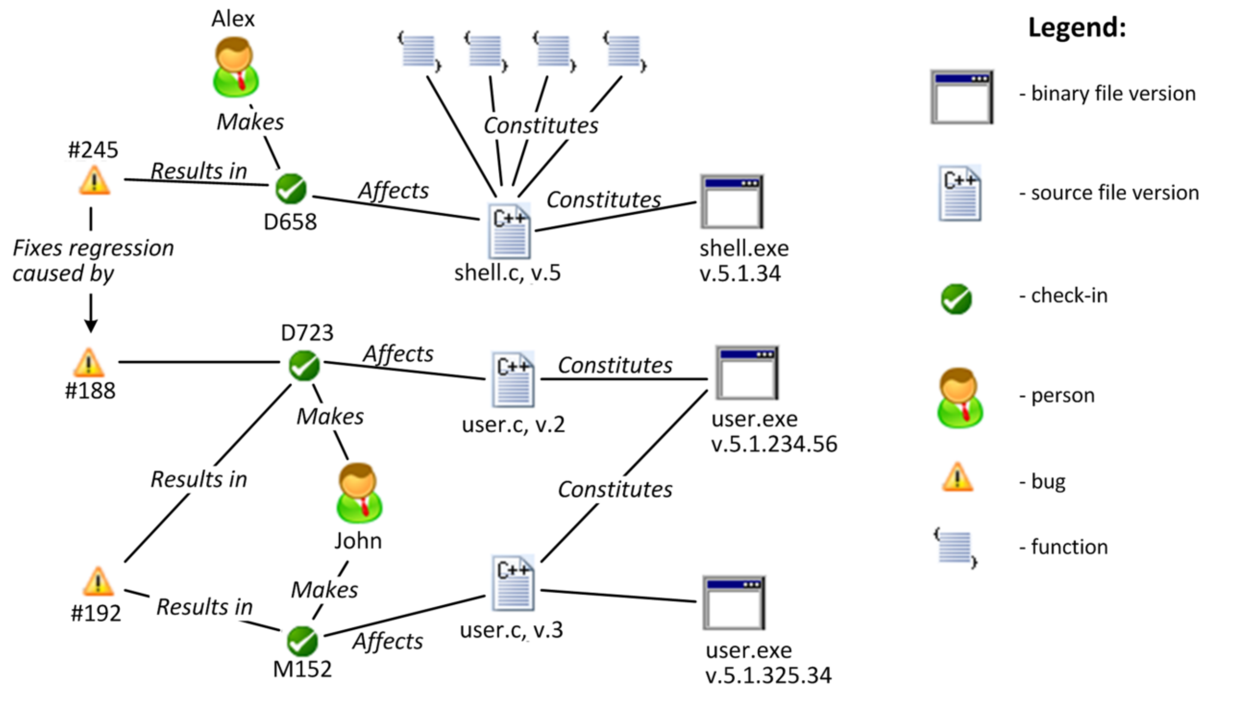

Figure 1. Entities and relations in the software system

developer along with other textual or numerical data, there is no strict format that the check-in description field must follow. Moreover, most VCSs have no mechanisms to enforce the developer to enter the valid bug ID, and, as a result, this important piece of information can be missing.

Despite all these difficulties reconstructing relations between entities in the software system is still possible. This is done using mining tools that are analyzing various sources of information about the software system, such as VCS, BTD, system's source code or mailing archives.

Mining tools have a number of important applications. One of them is collecting historical information on evolution of the software system. This information is used for many purposes, such as building statistical models that predict risks in that system. One kind of these models identify system's components with high risk of failure [4][5], another models identify code changes with high risk of introducing software regressions[6][7]. These predictions allow allocating testing resources more optimally, e.g. provide additional testing for the most risky components or fixes and, at the same time, reduce testing for those entities that have minimal risk.

Mining tools can be also used to collect information on a current state of the system. For example, they are used to analyze periodic builds of the system and determine what bugs are fixed in each particular build and apply predictive models to predict risk of these fixes [15]. They can also be used to build recommender systems such as Hipikat [10]. For a full survey of mining tools we refer to Kagdi et al [23].

Unfortunately, when multiple code branches are used during the development and maintenance of the software system, mining information about changes becomes more complex because of code integrations. In particular, ***associating check-ins with bugs becomes more difficult.*** As we discussed earlier, when the bug fix results in the code change either the ID of the bug record is stored in the check-in description, or the bug record itself contains a reference to the check-in. However, when integrating code, version control systems typically copy only the changes themselves but not the meta-information about the changes from the source branch to the target branch. In our example, the description of the check-in in the source branch might not be necessary copied to the target branch during the integration, especially when results of multiple code changes are integrated at once. Thus knowledge about who made a change, when, why, and which bugs were fixed is lost, unless both the source and target branch are analyzed. In this paper, we present an algorithm that facilitates this kind of analysis and we share our experience of using this algorithm at Microsoft.

4. Development and maintenance using multiple code branches

Code branching is a widely used methodology to streamline developer's work. It is used both during the development and maintenance of the software system. In this section, we describe common branching strategies that are used in professional software development.

4.1. Multiple code branches during maintenance

One reason to introduce code branches is improving reliability of the software system by separating code of its stable versions from code of new, unstable ones. After a stable version of the software system is released, developers have to start their work on its new version. However, at the same time they have to maintain the old version of the system. Maintenance activities include fixing reliability, performance and security issues [14] and result in numerous fixes in the system's source code. These fixes are wrapped into software updates and released to customers on a regular basis.

However, changes that occur during the development of the new version should not be included into these updates. Newly developed code has not undergone as much testing as the released old version of the system. As a result, it has lower quality and, thus, higher risk of software regression. Using the same code branch for both development and maintenance is practically impossible.

To solve this problem, developers typically create a separate code branch called a *development branch* that will be used to develop a new version of the software. The original version of the system's code will be used to develop updates for the stable version; it will become a *maintenance branch* (see Figure 1). Such schemas are used for example by the Apache and Eclipse open-source projects.

Another possibility is to create multiple maintenance branches, so every branch will be used to develop a different type of fixes. For example, one branch will contain only high applicability fixes (e.g. security or reliability fixes) that are shipped to every user of the product. Another branch will contain low-applicability fixes (e.g. performance improvements) intended to be downloaded only by customers who need these fixes. Such an approach ensures that every user has to install a minimal set of updates which, in turn, can minimize the impact of possible software regressions on the user community caused by low-applicability updates. In this case, however, all the changes in the high-applicability branch must be copied into the low-applicability branch as well. This is done through a typically automated process called *integration*, when changed files from one branch are integrated (copied) into the different branch (see Figure 2). The integration takes place in the form of check-ins on the destination branch.

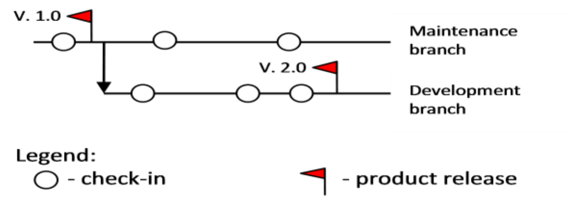

Figure 1. Using multiple code branches during software maintenance

Figure 1. Using multiple code branches during software maintenance

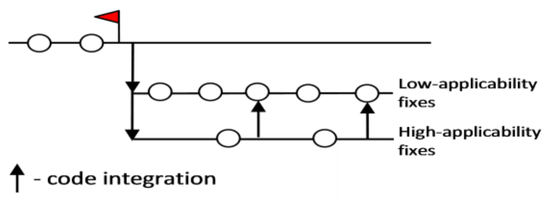

Figure 2. Using multiple code branches to reduce the risk of software regressions

Figure 2. Using multiple code branches to alleviate effects of software regressions.

4.2.Multiple branches during development

Another reason to introduce code branches is distributing development effort across multiple developers and development teams. In this case it can be beneficial to create multiple code branches, so-called *feature branches*. Every team that works on its own features of the system will maintain a separate copy of the system's source code in its private feature branch. This guarantees that new (and, potentially unstable) changes made by one developer do not affect another developers working on different parts of the system. Figure 4 illustrates the use of feature branches.

When development approaches a certain milestone, all the changes in feature branches *fb1*, *fb2*, *fb3* are merged back into the *main* (also called *trunk*) branch that is used to build and test the product. This process of copying changes from development branches into the trunk is called a *reverse integration*. Once all the reverse integrations are done, the product is tested. During testing some bugs can be found and corresponding fixes will be made either in the main trunk branch or directly in the development branch.

After a predefined quality criterion is met, all the code from the trunk branch is copied back into feature branches. Such a process is called a *forward integration*. Forward integration ensures that developers will use the newest version of the system when they will start working on its next milestone.

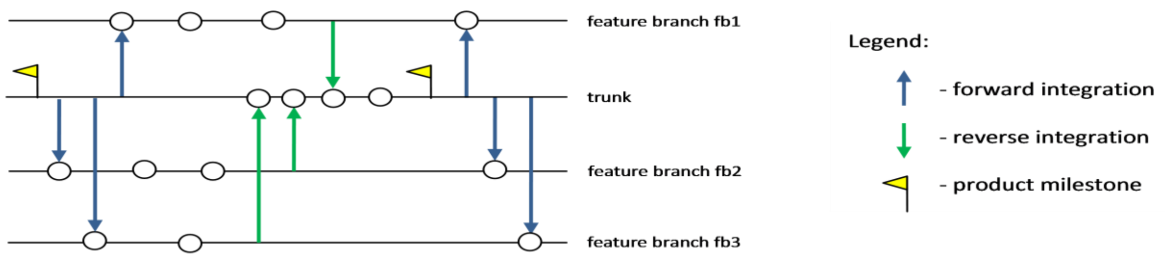

Figure 3. Multiple code branches during development

Figure 4. Multiple code branches during development

5. Mining multiple branches

Above we discussed two basic scenarios of using code branches. During software development and maintenance large projects with a long development history, such as Microsoft Windows, might use more complex branching configurations that combine both scenarios described above. On the one hand, new features are developed, which necessitates the development branch (or branches). On the other hand, all reliability, security, and performance issues fixed in the old version of the system must be also fixed in its new version. Thus any fix made in the maintenance branch must be also integrated in the development branch.

Such complex branching schemas provide benefits for development, but complicate mining software repositories. While analyzing the history of changes in the system we naturally have to concentrate on changes in the trunk branch because this is the branch used to build the final version of the product. But depending on the branching schema, the vast majority of code changes in the trunk can be just results of integrations from the development or maintenance branches. This poses a serious problem during the analysis of a software system. Any integration in the trunk branch can be a result of multiple changes in the development or maintenance branches; however, it is not possible to distinguish these changes by analyzing only the trunk. Many important pieces of information about the evolution of the system are lost:

- Information about **bugs**. Integration check-ins usually do not contain information about associated bugs. Such data can be retrieved only by analyzing check-ins in the development or maintenance branch, where check-ins are related to bug fixes. But even if the integration check-in does contain a reference to corresponding bug records, this information is hardly useful because one single integration check-in in the trunk usually contains multiple fixes, and thus references to multiple bugs. In order to reliably identify code changes that fix a particular bug, every particular change in the development branch must be analyzed;
- Information about the **developer who made the change**. Developers are checking in fixes into the maintenance or development branches, not into the trunk. Thus to get the name of the developer who implemented a change, the development or maintenance branches have to be analyzed. Analysis of fixes in the trunk branch will return only the names of the people who integrated changes into the trunk branch;
- Information about **code churn**. Source files or functions can be changed multiple times in the maintenance branch before being integrated into the trunk. If only the trunk branch is analyzed, just a single change in the trunk branch will be counted and the information of changes in the other branches is lost. Moreover, information on the time when these changes were made is lost as well;

Figure 5 provides an example of such information loss. Here the integration check-in that created version v.5 of some file in the trunk branch took place in August, but it contains numerous changes that occurred in development branch *Dev* and maintenance branch *Maint* during the May-July time interval. Without explicitly finding which changes were integrated, it is impossible to accurately determine the names of developers, time intervals, and bugs associated with these changes.

Moreover, we cannot assume that all the changes in maintenance or development branches are eventually integrated into the trunk. In fact, some recent changes in the development branch might not be integrated into the trunk yet, and some changes in the maintenance branch might not be integrated at all (for example because these components of the system would undergo a complete redesign in its new version). Thus mining only development and maintenance branches separately will not give us an accurate knowledge of the system; instead all branches have to be mined together.

Below we present an ***integration resolution algorithm*** that tracks integrations in the code of the software system. For each check-in in the trunk branch it determines changes in the development branch that were integrated into that check-in. We first describe the algorithm in a scenario with two branches and extend it then to a multi-branch environment.

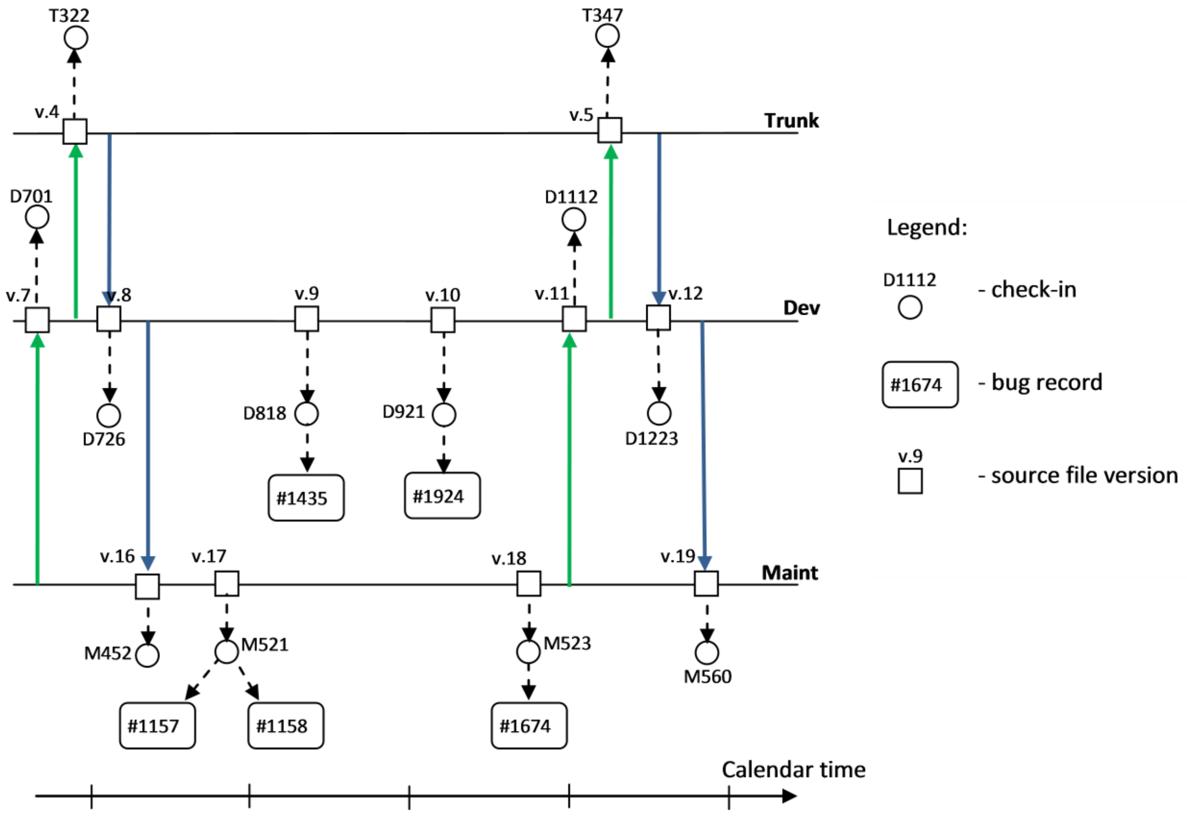

Figure 5. Histories of changes in the source file across multiple code branches

5.1. Integration resolution algorithm

The algorithm accepts the check-in c_t (the subscript here denotes affiliation of the entity to the branch) in the trunk branch *t* that was created as a result of integration from development branch *d*. The check-in c_t affects a set of source files F_t in the trunk branch. For every single source file $f_t \in F_t$ a number of versions $f_t[j], j \in (1,\ldots,M)$ exist, where the maximum version number *M* is different for every source file $f_t \in F_t$. These versions represent the evolution of the source file *f* in the branch *t*. Correspondingly, as a result of check-in c_t a new version is created for each of these source files.

The overall schema of the algorithm in Figure 6 is simple:

1. For the check-in c_t find the set of source files F_d in the development branch *d* that were integrated into the trunk branch;
2. For every source file $f_d \in F_d$, determine the time interval that contains all the changes in the branch *d* that were integrated into the trunk. This time interval is defined by a lower and an upper bounds;
3. Retrieve these changes and form a result set C_d^{all}.

The first step is trivial when the version control system records the fact of code integrations and the corresponding source branches. In fact, our algorithm enumerates all the source files in F_t. For every source file $f_t \in F_t$ in the trunk branch, the integration resolution algorithm retrieves a source of integration – the version $f_d[u]$ of the file f_d in the development branch. In our case this information is provided by the version control system. For version archives that do not record integration events (such as CVS) the algorithm proposed by Fischer et al. [13] can be used to identify the source of integration.

```
function ResolveIntegration($c_t$)
inputs: $c_t$, a check-in in the trunk branch that is a result of integration from the development branch
returns: $C_d^{all}$, a set of all check-ins in the development branch that were integrated into the trunk

$C_d^{all}$ ← {};
for each source file $f_t$ affected by $c_t$                /* Step 1: Enumerate sources of integration*/
    $f_d$_source ← source of integration for $f_t$ in the development branch;
    $f_d$ ← array of all versions of the file $f_d$     /* Step 2: Determine time interval (lower and upper border)*/
    sort $f_d$ ascending, so the latest versions of $f_d$ are last in the array;
    reachedUpperBorder = false;
    for j = number of versions in $f_d$ for file $f$ to 1 step -1
        $f_d[j]$ ← j-th element of $f_d$;
        if $f_d[j]$ is equal to $f_d$_source then reachedUpperBorder = true;
        if reachedUpperBorder then
            if ($f_d[j]$ is a result of reverse integration from the trunk to the dev branch) OR
               ($f_d[j]$ is integrated from the dev to the trunk branch)
                break;    // We reached the "lower border", exit the loop
            /* Step 3: Build result set with check-ins during the time interval*/
            $c_d$ ← check-in in the dev branch that caused change in $f_d[j]$;
            $C_d^{all}$ ← $C_d^{all} \cup c_d$                // Add check-in to the list of resulting check-ins
```

Figure 6. Pseudo code of the integration resolution algorithm

During **the second step**, our algorithm determines which versions of each source file f_d were integrated into the trunk branch. For this, it finds the lower and the upper bounds of the time interval that contains these versions. The upper bound is simply the source of integration under study, that is the version $f_d[u]$ of the source file f_d. The lower bound $f_d[l]$ is the version of the file f_d that was created by the previous integration between the trunk branch and the development branch.

At the ***lower bound***, contents of the files f_t and f_d are identical because all the previous changes in the trunk were copied into the development branch. Normally this copying is performed during the *forward integration*, when data from the trunk are copied back to the development branch. After this integration the file f_d starts changing in the development branch because of subsequent check-ins; thus contents of the files f_t and f_d are becoming different again. With the next *reverse integration* all these changes are brought back into the trunk branch and form the check-in c_t. This integration denotes an ***upper bound***, where contents of the files f_t and f_d are identical once again. Thus the reverse integration contains all the changes that occurred in the development branch between the lower bound (non-inclusive) and the upper bound (inclusive).

As we see, reverse and forward integrations play the role of a *synchronization mechanism* that synchronizes the file's contents in different branches. The integration resolution algorithm detects when the last synchronization is performed (a "lower bound") and collects all the changes up to the reverse integration under study (an "upper bound").

Figure 5 illustrates the situation described above. Here we are analyzing check-in *T347*, which is a result of reverse integration from the development branch into the trunk. Check-in *T347* affects file f_t and, as a result, version v.5 of this file is created in the trunk branch. In this case the upper bound is version v.11 of the file f_d in the dev branch; it is the last version of the source file f_d prior to the reverse integration. The lower bound is version v.8 created by the latest forward integration from the trunk to the development branch (check-in *D726*). All changes made to the source file between these two versions v.8 (excluded) and v.11 (included) are integrated into the trunk.

Another common integration scenario is depicted on Figure 7. In this case contents of the file f_t in the trunk branch are frequently copied into the development branch to keep source code in these branches synchronized in a long term (check-ins D45, D46, D47 on the figure). However, after a subsequent integration contents of the file f_d in the

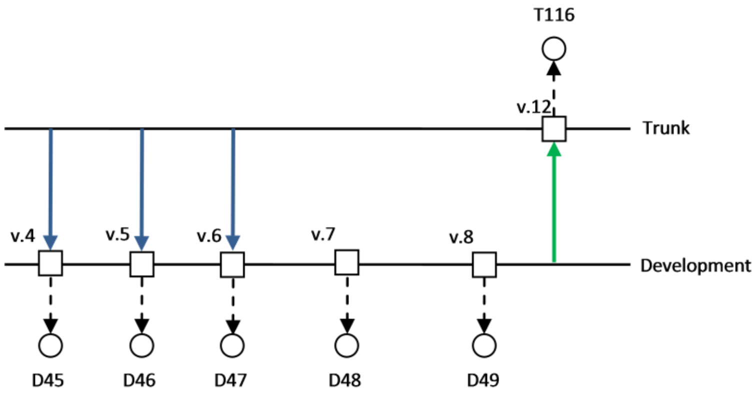

Figure 7. Processing of multiple forward integrations with integration resolution algorithm

development branch are modified and brought to the trunk using a reverse integration, which results in creation of version v.12 of the file f_t in the trunk branch. Here we analyze the check-in T116 in the trunk. In this case the upper bound is the version v.8 of the source file f_d in the development branch. The lower bound is the version v.6 that corresponds to the last forward integration from the trunk branch. As a result, the algorithm will correctly report that changes in versions v.7 and v.8 of the file f_d are integrated into the trunk branch.

It is important to notice that not only the latest forward integration from the trunk into the development branch can become a lower bound, but also any previous reverse integration from the development branch into the trunk can become a lower bound simply because it also leads to the synchronization of file contents between two branches. One example of such situation is shown on Figure 2, where changes in the high-applicability maintenance branch are frequently integrated into the low-applicability maintenance branch, but not vice versa.

The result of the second step is the sub-array $f_d\,(l,u]$ which contains all the versions of the source file f_d changed between the lower and upper bounds. This array can be associated with the destination of the integration, the file f_t, and allows to access information such as developers and bug fixes which are included in a code integration.

Finally, in the **third step** we setup links between the check-in c_t in the trunk branch and check-ins in the development branch that were integrated by c_t for file f_t. For every version of the file $f_d\,[j]$ with $j \in (l,u]$ we locate the check-in c_d that created that version. These check-ins constitute the set of check-ins C_d whose contents were later integrated into the trunk branch by check-in c_t.

The second and third steps are performed for each source file f_t in F_d affected by the check-in c_t. For every source file f_t we are forming its own set of check-ins C_d. Finally, the union of all sets $C_d^{all} = \cup(C_d)$ forms the result set C_d^{all} of all check-ins that were integrated into the trunk by the check-in c_t. (In Figure 6, the computation of C_d is omitted to reduce the complexity of the algorithm.)

Once the set of check-ins C_d^{all} is known, all the bugs, source files, and people related to a code integration c_t can be used by mining tools such as BCT [15] or Hipikat [10].

5.2. Finding lower bound for merge operations

Above we considered situations when the whole text of the source file is copied during the integration from one branch to another. However, such “force copy” integration is not always possible. For example, certain changes in the development branch were made before the forward integration and these changes must be preserved during that integration. Obviously, a “force copy” is not possible in this situation, so changes in the development branch are merged with the changes in the trunk.

Proper handling of merges requires some changes to be made to the integration resolution algorithm in order to determine a lower bound of integration properly.

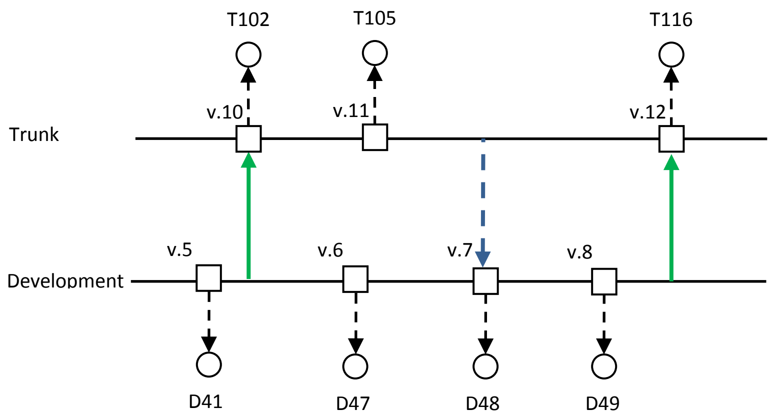

Figure 8. Integrations using code merge

Figure 8 provides an example of code merge. Here two changes were made to the trunk and development branches almost simultaneously: these are check-in T105 in the trunk branch (results in creating of version v.11 of the source file) and check-in D47 in the development branch (results in version v.6). In order to preserve both changes v.11 is merged, not copied into the development branch. This results in the check-in D48 (merge is shown by the dashed line). Finally, after the change D49 all the changes in the development branch are brought back to the trunk by the reverse integration. This results in the check-in T116 in the trunk branch which we want to resolve.

In order to handle such cases properly, we have to treat every code merge from the trunk to the development branch as a regular check-in, i.e., when we encounter a merge, we have to continue searching for a lower bound. This will result into minimal changes in the algorithm outlined in the Figure 6:

- The algorithm has to examine every check-in to determine if it is a copy or merge operation. Ideally the VCS should provide such functionality. If it does not, then the contents of the file version in the source branch before the potential merge and in the destination branch after the potential merge must be compared. If the contents of these file versions are the same, then a force copy occurred. Otherwise, this version was created as a merge and should be treated as a regular check-in.

  An example of such situation is depicted at the Figure 8. Here the contents of v.5 of the file in the development branch are equal to the contents of v.10 of that file in the trunk because there was a force copy between these files caused by reverse integration. Instead, v.7 of the source file in the development branch is different from the v.11 of the same file in the trunk because v.7 contains changes from both v.11 in the trunk and v.6 in the development branch.

- If the change is a merge, it should be considered by the integration resolution algorithm as a regular check-in and the search for a lower bound should continue. If the change is a force copy integration, the algorithm has found a lower bound.

5.3.Multi-stage integrations

So far we considered the scenario with only two branches, namely trunk and development. However, in practice ***multi-stage integrations*** are common, that is, changes are integrated through multiple branches before they reach the trunk. For example, Figure 5 depicts a scenario where check-ins made to the maintenance branch are integrated into the development branch and, finally, are integrated into the trunk branch. Another situation where multi-stage integrations are likely is when two maintenance branches are used for different kinds of fixes, as shown on Figure 2.

In order to track multi-stage code integrations we apply our algorithm ***recursively***. For every check-in c_t in the trunk we are getting a set of corresponding check-ins C_d in the development branch. If one of these check-ins $c_d \in C_d$ is a result of code integration from a different branch, we apply the integration resolution algorithm to c_d. Please note that here both force copies and merges can be considered as code integrations.

```
function ResolveMultibranch($c_t$, $branchesTraversed$)
inputs: $c_t$, a check-in in the trunk branch that is a result of integration from the dev branch
        $branchesTraversed$:  a set of all the branches encountered so far
returns: $C^{allbranches}$, a set of all check-ins in the development branches that were integrated into the trunk

$C^{\text{allbranches}} \leftarrow \{\}$
$b_t \leftarrow$ the name of the branch $c_t$ belongs to
$branchesTraversed \leftarrow b_t$
$c_t.intSources$ = ResolveIntegration($c_t$)
$C^{\text{allbranches}} \leftarrow C^{\text{allbranches}} \cup c_t$.intSources
for each check-in $c_d$ in $c_t.intSources$
    $b_d \leftarrow$ the name of the branch $c_d$ belongs to
    if $c_d$ is a result of integration and $branchesTraversed$ does not contains $b_d$ then
        $c_d$.intSources = ResolveMultibranch($c_d$, $branchesTraversed$)
        $C^{\text{allbranches}} \leftarrow C^{\text{allbranches}} \cup c_d$.intSources
```

Figure 9. Pseudo code of the multistage integration resolution algorithm

The pseudo code of the algorithm is shown in the Figure 9. Here every check-in is represented by an instance of the corresponding class. Its *intSources* field contains the list of check-ins from the lower-level branches that were identified with our resolution integration algorithm.

Please note that proper handling of merges require special care. On the one hand, merge can be seen as a special case of code integration and thus should be properly resolved. On the other hand, merges can form loops with other integrations. One example of such a loop can be seen at the Figure 8. Here we are resolving integration from the development branch to the trunk (check-in T116). However, one of the check-ins in the development branch itself is the integration from the trunk (D48) and should not be resolved. To prevent the algorithm from resolving such circular integrations the multistage algorithm maintains a set of the branches it already went through. If the multistage algorithm detects that certain check-in leads to a circular integration, it skips that check-in from the further analysis.

The algorithm represents the history of check-ins and integrations with a tree structure (see Figure 10). The nodes represent check-ins and edges between these nodes represent integrations. *Leaf nodes* correspond to *plain fixes*, namely, to check-ins that are not results of code integration from other branches. In most cases, these fixes can be related to bugs in the bug-tracking database. *Inner nodes* represent check-ins that resulted from code integration. Edges point from the code integration to its sources and represent the flow of integrations. For example, Figure 10 shows the tree of code integrations that corresponds to the multi-branch scenario shown on Figure 5. Here check-in *T347* subsumes five other code integrations and contains ultimately the fixes #1435, 1924, 1157, 1158, and 1674.

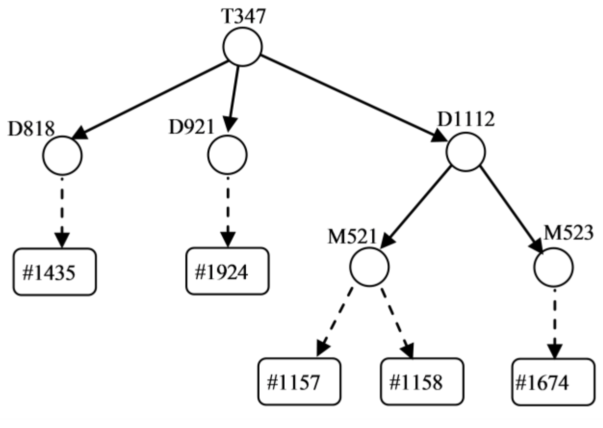

Figure 10. Representing code integrations in form of a tree

The tree allows us to quickly find the source and destination of code integrations for any check-in. For example, to find which check-ins contributed to check-in c_t, we lookup c_t in the tree, traverse its sub-tree, and collect all the leaf nodes. To facilitate search references to check-ins, the tree can be indexed or stored in a hash table.

Once we have the leaf nodes, we can find any information related to the changes that are contained in the code integration. This includes entities, such as bugs, file versions, people, and binaries, which are directly or indirectly related to check-ins.

5.4.Limitations of the algorithm

Our algorithm has the assumptions that all integrations between branches are carried out in regular patterns. It assumes that for every reverse integration that occurs between a pair of branches there is a corresponding forward integration. In other words, fixes can flow between development and trunk branches, between maintenance and development branches, but not between maintenance and trunk branches (see Figure 5).

This assumption is true for most systems with a well-established development processes (e.g. Vista SP2). In fact, it represents the normal flow of development of the software system and thus is reasonable. However, in some cases code integrations might not follow such regular patterns. For example, reverse integrations might flow from maintenance to development branch and from development to trunk, while the forward integrations might be made directly to both branches (see Figure 11). Another problem is circular code integration, when code from maintenance branch is integrated into the development branch, from development branch into the trunk, and from trunk back into the maintenance branch.

In the cases described above, our integration resolution algorithm will not be able to locate the corresponding lower bound and might produce wrong results. In our case such non-standard situations were rare. In fact, they constitute only 2% of all the integrations considered in this study, which proves soundness of our assumptions. However, non-standard situations might occur more frequently in other projects.

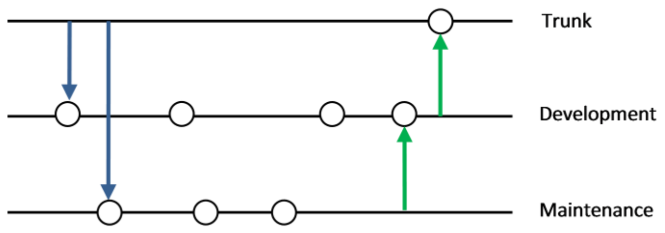

Figure 11. A non-standard code integration

6. Practical results

We used the integration resolution algorithm during the Windows Vista Service Pack 2 (SP2) development to determine which bugs were fixed in periodic builds of the system. This, in turn, allowed us to predict the risk of software regressions for each of these bugs. It is important to emphasize that with the term “bugs” we refer to bug records in the BTD, which besides actual faults can also include reliability, performance, and usability improvements.

Furthermore, with information collected with our algorithm we were able to define a number of code metrics that characterize code integrations. These metrics can be used to monitor the development process or as predictors in risk prediction models.

6.1.Collecting information on bug fixes

In our earlier work, we developed a statistical model to predict risk of software regressions in Windows software updates based on the properties of these updates [7], such as the number of source files and functions affected by the corresponding code change and properties of the corresponding bug (is this update a new functionality or a bug fix). The predictions were used to optimize the testing process: it was recommended that updates with a high predicted risk of software regressions should be tested more thoroughly. The model proved to be successful and, naturally, we wanted to use it to predict risk of regressions during the Vista SP 2 development.

However, during the work on the Windows Vista SP2, a complex branching schema was employed. The actual development was performed in multiple development and maintenance branches and then all the changes were copied into a trunk branch. Multi-staged code integrations between these branches were common; some changes travelled through 2 or 3 different branches before being integrated into the trunk.

The trunk branch was used to produce regular builds of the product that were released for testing. To apply our statistical model to Windows Vista SP2, we had to determine which bug fixes were integrated into the particular build and determine properties of these fixes. To do this, we implemented the integration resolution algorithm on top of our tool for data collection, called *Binary Change Tracer (BCT)* [15]. The whole process of data collection is performed in three stages.

During the **first stage**, BCT accepts two sets of binaries that belong to the previous and current builds and compares these binaries one by one. For every pair of binaries (old and new ones) the tool uses symbol files to retrieve a complete list of source file versions used to build these binaries. Then BCT turns to the version control system to retrieve a history of code changes in every source file in that list. Next, BCT associates code changes in the source files with the corresponding bugs in the bug-tracking database and saves all the mined information into its own BCT database. Once all the binaries are processed, full information about changes between previous and current builds is accumulated.

In addition to information about code changes, our new version of BCT now also collects information about which of these changes are code integrations and from which branch they were integrated from (their source of integration). This information is needed to run our code integration algorithm in the third stage. For every file that was modified because of code integration, a link to the integration source is stored in the BCT database. As a result, information about code integrations between these branches is also accumulated in the database.

After data collection, the **second stage** begins: information about changes in the trunk branch between two builds is loaded from the database into the computer memory, where it is represented in the form of the object model. Every source file, source file version, check-in, person, and bug record is represented through an instance of the corresponding class. Relations between these entities are represented as references to other objects. Every object representing a source file version has a link to the source file object (which is a container for all versions of that source file), every source file version has a link to a related check-in object, and, finally, check-ins that can be associated with bugs have links to corresponding bug objects. Furthermore, if a file version was created as a result of code integration, a link to the sources of the integration is provided.

During the **third stage**, we launch our multi-stage integration resolution algorithm for all integration check-ins found in the trunk branch. It retrieves data on code integrations from the BCT database and incorporates them into the object model. As a result, for every check-in in the trunk branch an integration tree is created. This allows us to find a full list of bug fixes that were brought to the trunk.

Finally, we compute code metrics for every bug record. These metrics include number of source files changed to fix the bug, number of code lines affected, and if this bug record is a new functionality or bug fix. With this information, a risk of regression for every bug is predicted using a logistic regression model as shown in our earlier work [15].

During the final stages of SP2 development data on code changes was collected once or twice a week. A report was presented through a web-interface; it contained the list of fixes in the build and their risk of regression. Information on risk of changes helped test engineers to decide on how much testing was necessary for a fix. Moreover, knowing a full list of bugs and code changes in the build helped project management to better track the progress of the project.

To speed up the data collection, we used multiple copies of BCT running on different machines, so that they could simultaneously process builds coming from different code branches. The first stage took 12-24 hours to collect data (19 hours average), and the second stage normally took 25-50 seconds (35 seconds average) to load data on changes

Table 1. Proportion of Vista SP2 bug fixes in different branches

Time	Percentage of bug fixes in different branches		
	Trunk	1st Level	2nd Level or more
Oct 2008	12.8%	33.3%	53.9%
Nov 2008	8.0%	34.5%	57.5%
Dec 2008	4.0%	50.7%	45.3%
Jan 2009	0.5%	49.5%	50.0%
Feb 2009	26.1%	27.5%	46.4%
Mar 2009	9.6%	16.7%	73.7%

in a trunk branch from the database. The third stage, which runs the integration resolution algorithm to build an integration tree for every check-in and calculates risk of regression for every bug, took 1-40 seconds (7 seconds average). All the measurements were done on a Pentium-D 3.4 GHz machine with 2 GB RAM.

We also used our integration resolution algorithm to compute the fraction of Vista SP2 bugs that were fixed directly in the trunk branch, bug fixes for which it took one step to get integrated into the trunk (1st level of integration), and bug fixes for which it took more than one step to get integrated into the trunk (2nd or higher level of integration). We collected this data during the last six months of the Vista SP2 history, beginning October 1, 2008 and ending March 31, 2009 (which was close to Vista SP2 release date April 28, 2009). This time span included important milestones of the SP2 project, such as Beta and Release Candidate.

The results in Table I clearly show that only a small fraction of all the check-ins in the trunk branch can be associated directly with the bug records in the bug-tracking database. The vast majority of the fixes, instead, were done in the maintenance and development branches and then integrated into the trunk. Moreover, most fixes required two or more integrations before finally being included into the trunk branch. Without using our multi-stage integration resolution algorithm these bugs could not be properly associated with check-ins in the trunk branch. This, in turn, would cause a major loss of information about changes in the system.

The high percentages Table I for 1st and 2nd level integrations in Table I demonstrate the ***importance of dealing with branches when mining software repositories***.

6.2. Collecting integration metrics

In addition to collecting data, the integration resolution algorithm allows us to define a number of metrics that describe integrations in the project. The metrics are defined for either development or trunk branches. Here we assume that during the integration a set of check-ins $\{c_d^1, \dots, c_d^N\} = C_d^{all}$ are integrated from the development branch(es) into the trunk branch. As a result check-in c_t is created in the trunk.

We introduce the metrics on the level of check-ins which is useful for prediction models. To track the progress of a project, the metrics should be aggregated with *average* and *maximum* operations and observed over time.

Integration time (ITIME). This metric is defined for check-ins in the development branch. It shows how much time is required for a check-in to reach the trunk:

$$ITIME(c_d) = Creation\ time(c_t) - Creation\ time\ (c_d)$$

Increasing values of the ITIME metric can signal a slowdown in the process of integrating changes into the trunk, which, in turn, delays testing of these changes.

Number of Integrated Check-ins (NIC). This metric is defined for code integrations in the trunk branch. The value of the metric is equal to the number of check-ins brought into the trunk by a single integration:

$$NIC(c_t) = \left|C_d^{all}\right|$$

The NIC metric can be also viewed as the total number of nodes in the integration tree, excluding the root node. Correspondingly, we can define the number of integrated leaf check-ins (**NILC**) metric as a number of plain fixes integrated into the trunk, and the number of non-leaf check-ins (**NINLC**) metric as a number of interim integrations.

The NIC, NILC, and NINLC metrics represent size and complexity of the integration.

Integration depth (ID). This metric is defined for check-ins in development branches. Its value is equal to the number of interim branches the check-in has to traverse before it reaches the trunk. Since in our case a very strict set of integration rules was used, the value of metric was constant for all the fixes coming from the same branch. However, this metrics will be useful for projects with less strict integration schemas.

Number of Delayed or Missing File Integrations (NDMFI). Suppose that the check-in c_d affects a set of source files F_d. After integrating contents of c_d into the trunk, the check-in c_t that affects a set of source files F_t was created. Since integrations are performed for source files separately, there is the possibility that some source files of F_d were not integrated into the trunk. These source files form a subset $F_d^{missing} = F_d - F_t$ of missing (they might have not been integrated by accident or discarded) or delayed (they might be integrated later) file integrations. The value NMDFI for check-in c_d is calculated as a cardinality of the $F_d^{misings}$ set*:*

$$NDMFI(c_d) = |F_d^{missing}| = |F_d - F_t|$$

Similarly, we can define the NMDFI metric for a check-in c_t in the trunk branch which changes files F_t. Here F_d^{all} is the set of source files affected by all check-ins C_d^{all} that were integrated into the trunk by c_t. Now $F_t^{missing} = F_d^{all} - F_t$, and, correspondingly

$$NMDFI(c_t) = |F_t^{missing}| = |F_d^{all} - F_t|$$

Since all the files in c_d were changed in the development branch, it can be reasonably assumed that they should be also changed in the trunk branch as well. If some of these files are not integrated into the trunk, this can lead to the potential error. Even if integration of the particular source file is delayed, it might indicate potential problems, or, at least, is a deviation from the standard integration procedure. In any case, this metric can be an interesting predictor of future-fault proneness of a software system.

Given incomplete integrations, one can think of a recommender system that issues a warning whenever incomplete integration happens. Such a system is easy to build with our integration resolution algorithm and can help to avoid errors caused by incomplete integrations.

7. Conclusion

In the presented paper we addressed the problem of mining software repositories in the presence of multiple code branches for development and maintenance. We described common branching schemas and introduced an algorithm to resolve integrations across multiple branches. Our algorithm tracks the flow of code integrations and matches code changes in a trunk branch to corresponding changes in other branches.

The algorithm proved to be extremely valuable during the development of Vista SP2. It allowed us to accurately determine which bugs were fixed in every build of the system and predict risk of software regressions for these bugs. In turn, this information allowed better control over the development process and optimize testing of builds.

Our algorithm has a number of assumptions on the data provided by the BTD and VCS. We relied on the VCS as on the source of information on integrations; however different VCS implementations might not provide this data. In this case the algorithm described by Fischer et al. [13] can be used to infer this information. Another assumption on the strict order of integrations can be more severe. Although in Vista SP2 only a small fraction of integrations did not follow that order, other systems might have less established integration schedules. Thus we plan to develop an extended version of the integration resolution algorithm without any assumptions regarding the flow of integrations.

We implemented our integration resolution algorithm as a stand-alone tool. Although functionality similar to the presented algorithm could be implemented directly in the VCS, this will not solve all the problems associated with integration resolution.

First, implementing the integration resolution algorithm as a part of the VCS will not allow mining the vast amounts of historical data for legacy systems. Such data can be particularly useful for building predictive models, such as regression prediction model [7]. In order to be trained, such models require vast amounts of historical data on bugs and corresponding code changes, which can be impossible to mine without integration resolution algorithm. Also integration data might be helpful to monitor development of the software system. It is possible that certain metrics, e.g. integration time (**ITIME**) or number of missing integrations (**NDMFI)** could be used as health indicators of the system's development. Tracking how these metrics change over time might uncover important trends in the development process.

Second, adding integration resolution functionality to the VCS itself might not be always possible. For example, the source code of the VCS might not be available, or the VCS can be maintained by a different team of developers who are unable (or unwilling) to make changes into the system. Thus our implementation of the algorithm as an extension of the BCT mining tool provides better flexibility than embedding it directly into the VCS.

Nevertheless, a number of improvements can be made in the existing design. For example, launching a tool to collect data separately for every code branch is both inefficient and inconvenient, especially for projects with many code branches. To solve this problem integration resolution algorithm must be implemented in the BCT itself. The algorithm will then be launched when the trunk branch is processed and will not require separate processing of other branches. Another way to improve performance would be parallelizing the data collection operations performed by the BCT tool, which would result in performance improvements on multi-core systems.

8. References

[1] Vliet, Hans Van. *Software Engineering: Principles and Practices.* West Sussex, England : John Wiley & Sons, 2000.

[2] X. Ren , F. Shah , F. Tip , B. Ryder , O. Chesley, Chianti: a tool for change impact analysis of java programs, Proc. of 19th conference on Object-oriented programming, systems, languages and applications, 2004, Vancouver, BC, Canada

[3] J. Jones , M. J. Harrold , J. Stasko, Visualization of test information to assist fault localization, Proc. of the 24th International Conference on Software Engineering (ICSE 02) , 2002, Orlando, FL

[4] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," Proc. 27th Int'l Conf. Software Eng. (ICSE 05), 2005, pp. 284–292.

[5] T. Zimmermann, N. Nagappan, Predicting Subsystem Defects using Dependency Graph Complexities, Proc. of 19th International Symposium on Software Reliability Engineering, Trollhattan, Sweden, 2007.

[6] A. Mockus and D. Weiss, "Predicting Risk of Software Changes," Bell Labs Tech J., vol. 5, no. 2, 2000, pp. 169–180.

[7] A. Tarvo, Using Statistical Models to Predict Software Regressions. Proc. of 20th Intl Symposium on software reliability engineering, 2008, Redmond, WA

[8] Appleton, B., Berczuk, S.P., Cabrera, R., Orenstein, R.: Streamed lines: Branching patterns for parallel software development. In: Proceedings of PloP' 98, published as TR #WUCS-98-25, Washington Univ., Dept. of Computer Science (1998)

[9] J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" Proc. 2nd Int'l Workshop Mining Software Repositories, 2005, pp. 24–28.

[10] D. Cubranic and G.C. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," Proc. 25th Int'l Conf. Software Eng. (ICSE 03), 2003, pp. 408–418.

[11] D. Perry , H. Siy , L. Votta, Parallel changes in large-scale software development: an observational case study, ACM Transactions on Software Engineering and Methodology, v.10 n.3, p.308-337, July 2001

[12] T. Zimmermann and P. Weißgerber , "Preprocessing CVS Data For Fine-Grained Analysis," *Proc. Mining Software Repositories,* pp. 2-6, 2004

[13] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," Proc. Int'l Conf. Software Maintenance (ICSM 03), 2003, pp. 23–32.

[14] IEEE Standard for Software Maintenance. 1998. IEEE Std 1219-1998.

[15] A. Tarvo, Mining Software History to Improve Software Maintenance Quality: A Case Study. IEEE Software 26(1): 34-40 (2009)

[16] S. Kim, T. Zimmermann, K. Pan, E. J. Whitehead Jr., "Automatic Identification of Bug-Introducing Changes," Proc. Int'l Conf. Automated Software Engineering (ASE 2006), pp. 81-90

[17] S. Kim, E. J. Whitehead Jr., Y. Zhang: Classifying Software Changes: Clean or Buggy? IEEE Trans. Software Eng. 34(2): 181-196 (2008)

[18] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, P. T. Devanbu, "The promises and perils of mining git", Proc. Int'l Work. Conf. Mining Software Repositories (MSR 2009), pp. 1-10

[19] T. Zimmermann, "Mining Workspace Updates in CVS", Proc. Int'l Workshop Mining Software Repositories (MSR 2007)

[20] M. Kim, D. Notkin, "Program element matching for multi-version program analyses", Proc. Int'l Workshop Mining Software Repositories (MSR 2006), pp. 58-64

[21] C. C. Williams, J. Spacco, "Branching and merging in the repository", Proc. Int'l Workshop Mining Software Repositories (MSR 2008), pp. 19-22

[22] T. Mens: A State-of-the-Art Survey on Software Merging. IEEE Trans. Software Eng. 28(5): 449-462 (2002)

[23] H. H. Kagdi, M. L. Collard, J. I. Maletic: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance 19(2): 77-131 (2007)