VERSION CONTROL INTEGRATION OF BUILD MAINTENANCE TOOLS WITH FORMIGA

by

Reed Johnson

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the Degree of

Master of Science
in Computer Science

at
The University of Wisconsin-Milwaukee
May 2015

ABSTRACT

VERSION CONTROL INTEGRATION OF BUILD MAINTENANCE TOOLS WITH FORMIGA

by

Reed Johnson

The University of Wisconsin-Milwaukee, 2015
Under the Supervision of Professor Ethan V. Munson

The task of build maintenance consists of creating, configuring, and updating the build system of a software engineering project. A project of sufficient size and scope is likely to have some sort of build system due to the complexity and time required to create a finished product. Build maintenance has been shown to greatly increase the cost of developing software due to the common need to modify a build system at the same time as the source code. Unfortunately, there is little in the way of tool support to assist developers with build maintenance.

Formiga is a build maintenance and dependency discovery tool developed by Hardt [1]. Formiga provides support for build refactoring, dependency identification, and automatic build updating based on modifications to source code. This thesis expands upon the original Formiga tool by investigating what kind of hurdles would be involved in integrating it with a production-quality version control system. An initial implementation of version control integration is built on top of the Formiga IDE plugin. It makes use of a mock version control system to keep track of file and file dependency history. This work, while not integrating with a production-quality

version control system, lays a basis on which to perform that full integration in future iterations of Formiga.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

When working on a software project of any appreciable size, a build system can save hours of work that would be spent manually creating various kinds of output files. A build system can be used to automate the construction of libraries and executables, among other things, from source files and other resources. It can run test suites and create executables for multiple kinds of environmental configurations. While automation allows for a wide range of artifacts to be produced with relatively minimal effort to most developers, creating and modifying builds can be a time-consuming effort itself. As a build system adds different capabilities, it grows in complexity, which can make it difficult for the average developer to maintain. Unfortunately, regardless of complexity, build system maintenance simply cannot be ignored. Research has shown that as source code grows in size and complexity, so must the build system [2].

One of many difficulties associated with modifying a build system is understanding dependencies between tasks and dependencies between files. The size of a build system can make this worse as the tasks used to create a project artifact can be spread over hundreds or even thousands of lines of code. In addition, most build files use variables whose content may consist of metacharacters such as wild cards or regular expressions used to match a range of files. It can be difficult to completely understand all the metacharacters in a build system. The time required to understand the structure of a build system can discourage developers from updating it when needed and may lead them to introduce errors as they modify the build system.

Version control is also a necessity in any software project of sufficient size. A version control system allows multiple developers, potentially separated by significant distance, to work together on a single code base. Additionally, a version control system maintains a history of development work and allows developers or other stakeholders in a project to view or build a software project at multiple points in its development history.

When talking about version control systems there are two main styles that have to be considered. One of the two styles is known as Centralized Version Control Systems *(CVCS)*, such as CVS[3] and Subversion [4]. CVCS repositories are characterized byhaving a single canonical repository for each project. Any development done on a project stored in a CVCS is performed through a "checkout" taken from the repository which is essentially a snapshot of that repository (or a part of the

repository) at a given point in time [5]. In addition to working on a snapshot of a repository, another key characteristic is that write-access to a repository is usually restricted to a known set of developers. Write-access must be restricted because committing a change to the repository will update the canonical version of the project. Finally, a common practice when using CVCS is to designate and maintain a "main" branch, creating new branches based on the main branch as needed in order to release new versions of a piece of software [5].

The second type of version control system is known as a Distributed Version Control System *(DVCS)*, such as Git[6] or Mercurial [7]. As its name implies, the biggest difference between a DVCS and a CVCS is that repositories are *distributed*. When dealing with a DVCS, each checkout is itself a repository. That means that each checkout from a DVCS contains all of the included resources and the complete commit history. Write-access is not as critical with a DVCS because commits are made to a developers' own local repository and can later be merged with other repositories. A consequence is that, unlike a CVCS, there is no enforced "main" branch. Often times, a project will identify a canonical branch, but it is not the same as the canonical repository found in CVCS [5].

Formiga is a tool developed by Hardt in 2014 that visualizes build dependencies and assists developers with Ant build system maintenance. It is implemented as an Eclipse plugin which allows it to use workspace listeners to monitor a project and detect when changes have been made to a project or the project's build system. One of the major features of Formiga is that it provides refactoring capabilities similar

to that of Eclipse. When changes are made to a project (such as a file addition or removal), Formiga alerts the developer about which targets within the build system will be affected and how. Formiga calculates the file-to-file dependencies by simulating the Ant build process in memory. File dependencies can then be displayed in a directed graph where nodes represent files and the edges between them represent a generator-generatee relationship.

As difficult as it can be to understand the dependencies within a build system at any given time, it can be equally difficult to understand how dependencies change over the lifetime of a project. Formiga helps a developer understand the myriad dependencies in an Ant build system, but not their history. Because Formiga is only able to operate on a software project within a given workspace it can only provide information about a single version of a project. In order for Formiga to understand and visualize historical information about dependencies it would need to interact with a version control system. Build dependencies could be used to inform developers of what library versions are needed to run a given version of software. This information would be useful to developers who want to use or develop for a certain version of a software project. This thesis seeks to lay the groundwork for integrating Formiga with a production-quality version control system by capturing and storing dependency history.

The remainder of this thesis is organized as follows. Chapter 2 presents background information on some commercially available version control systems. Chapter 3 briefly discusses features of the original implementation of Formiga. Chapter

4 describes the implementation of the updated "Formiga" tool and how it captures and records dependency history. Chapter 5 discusses potential future improvements and Chapter 6 concludes the paper.

Chapter 2

Survey of Commercial Version Control Systems

This section describes the two major paradigms in version control systems. Section 2.1 provides a brief introduction to some terminology that is common to nearly all version control systems. Section 2.2 briefly discusses the history of centralized version control systems before going over some of their key characteristics. Section 2.3 gives a similar treatment to distributed version control systems, mentioning some of the factors that contributed to the birth of the popular DVCS Git and covering what makes distributed version control different from centralized version control.

2.1 Introduction to Version Control Ideas and Terminology

A key concept in all types of version control systems is that of the repository. The repository is a combination of a file structure, metadata, and some list of changes made to the contents of the repository's file structure. Most repositories have some internal model to represent the file structure they are storing and this internal file structure is used to perform operations that developers associate with version control systems such as merging and branching [8]. The files themselves are generally represented as a sequence of lines or a blob of bits when dealing with text files and binary files respectively. When something makes a change to one of the files being stored in the repository and records that change by making a commit, this modifies the internal model held by the repository.

Modifying the internal model of the repository is referred to as a "patch" or a "revision". The history maintained by a version control system is actually just a sequence of the patches or revisions that led to the current state of the repository's inner model. A developer can retrieve any previous version of a file because the repository can apply or reverse the patches necessary to return the repository to any desired version [8]. Adding a patch to a repository can also cause a conflict if the internal model assumed by the patch is different than the actual internal model of the repository. An example of such a situation would be if a patch deletes a file that doesn't exist in the current repository. Such conflicts can sometimes be merged

automatically, but often they require human intervention in order to be resolved [8].

Many version control systems also support some form of branching. A branch is itself a complete repository, but it is special because it is made as a snapshot of another repository at a given point in time. Though a branch is based off one repository, making changes in it will not affect the original repository [8]. Branches can also be merged into other repositories, which makes them especially useful for working on features without disrupting the "main" repository.

2.2 Centralized Version Control Systems

One of the first version control systems to be developed is known as Source Code Control System *(SCCS)*, which was developed at Bell Labs in the early 1970s [9]. In SCCS all files in the repository are stored on a central server, and developers access the repository by working on workstations that are connected to that central server. When a developer wants to make changes to a file then that file is locked on the server and no other developers can work on it until the original developer is finished and the lock is removed [8].

In the early 1980s the Revision Control System *(RCS)* was released by Tichy [10]. RCS's implementation is based on the Unix *diff* utility and stores a series of deltas rather than complete revisions. RCS stores revisions using a series of backwards deltas, whereas SCCS stores a series of forward deltas. Backwards deltas make retrieving the current state of the repository faster in RCS than in SCCS. Another

key difference is that SCCS stores two files for each file under version control: a file with the first version and a series of deltas, and a lock file. RCS stores the file locks within the version file. If a user were to leave a team using SCCS without releasing his file locks, then root privileges would be required to remove the lock files because they belonged to the user who set the locks.

Concurrent Versions System *(CVS)* gained popularity over SCCS and RCS and maintained popularity until the turn of the century. In a departure from SCCS, which was oriented to individual files, developers using CVS "check out" the current version of the entire repository from the central server which creates a local copy of the repository. Perhaps the biggest advantage of this system over SCCS is that more than one developer can make tentative changes to a file at a given time because each developer gets a complete writeable copy of the repository [8].

The Subversion *(SVN)* system was developed in 2000 in order to address some of the problems that developers had experienced with the CVS tool. Some improvements over CVS include using a database backend instead of a collection of RCS files, atomic commit operations, and branching and tagging as cheap operations [11]. Subversion does not provide a radical departure from CVS in terms of the underlying version control system. By the developers' own admission, they were not looking to radically change how the underlying systems worked, they just wanted a version control system without the bugs present in CVS. The Subversion project joined the Apache Software Foundation in 2010 and it remains one of the most popular open-source CVCSs [4].

The defining characteristic of a Centralized Version Control System *(CVCS)* is that it takes a client-server approach to revisions [12]. SCCS, CVS, and Subversion all clearly take a centralized approach to version control. A result of taking a client-server approach is that most CVCSs have a single canonical repository [5]. This single, central repository is the authority on the current state of the repository along with its history. An important benefit of this is that if for some reason code needs to be *completely* removed from the repository, such as when a project is legally required to remove some code, this is completely possible. If everyone had their own local copy of the complete repository, that might not be possible. Having a central authority means that leadership in a project can have very tight control over which developers can push changes to the main repository and over the changes they can make. This can work as double edged sword on a software project. As a positive, project leadership does not have to worry about unknown or untested developers pushing, for example, untested changes to the central repository. But a disadvantage is if the ratio of "approved committers" to "unapproved committers" is low, then the approved committers may spend a disproportionate amount of time verifying the quality of potential changes.

A second requirement of CVCSs is a network connection to the repository. When a developer checks out a snapshot of the repository, the only thing they get is a copy of the file system stored within the repository. If a developer wants to check the history of the repository, see comments about individual commits, or commit their own changes to the repository they must have a connection to the repository

server [13]. Developers can still work when they are without an Internet or intranet connection, but they are limited to programming.

All of these characteristics indicate that there are certain situations in which a CVCS may be the preferred method of version control for a development team. If it is important that access to the repository is tightly controlled, then a CVCS may be a good choice. If it is important that there is only one copy of the repository, for example if some code needs to be completely expunged from that repository for legal reasons, then a CVCS may be a good choice. If most developers working with a repository can be expected to have Internet or intranet access to the repository server, then a CVCS may be a good choice. No single one of these factors means a team has to use a CVCS, but they do indicate that using one may not be detrimental to development efforts.

2.3 Distributed Version Control Systems

Distributed version control systems *(DVCS)* have been growing in popularity since the mid 2000s. Mercurial, Bazaar[14], BitKeeper[15] and Git are examples of this newer paradigm for version control systems. Git itself was developed by Linus Torvald to aid in the development of the Linux kernel. Torvald began development of Git in 2005 because of his dissatisfaction with previous DVCSs available at the time –BitKeeper specifically[16]. Though dissatisfied with BitKeeper, he also felt strongly that CVCSs were a flawed approach to version control. He felt so strongly

about this that he stated if they had too many issues developing their own version control software he would sooner stick with BitKeeper than use a CVCS[16]. In particular, one of his larger issues with CVCSs were that he felt the "approved committer" model proved to be too big of a bottleneck on productivity. Thus, in 2005 Torvald began development on what would become one of the more popular DVCSs[16].

As its name implies, the defining characteristic of distributed version control systems is its distributed nature. Rather than taking a server-client approach, each developer that uses a DVCSs has a complete copy of the repository (history and all) on the local machine [5]. Because each developer has a complete local copy of the repository, there is no "main repository" in the same sense as a CVCS. Generally what happens instead is that developers choose one or more "principle" branches that are treated similarly to the master repository found in a CVCS [5]. By having no single canonical branch, developers have the ability to pick and choose which revisions or patches they want to merge into their local repository [13]. Not only can developers be selective about what they merge into their local repositories, they also have the ability to be selective about which of their committed revisions they push back to the "principle" branches. Furthermore, most DVCSs give developers the ability to choose how they present their changes to other developers by combining and splitting individual commits [17]. When used properly this capability allows developers to organize their changes by the issue solved or by some other logical structure.

The local repository removes the need for developers to be "approved committers" to get the benefit of using a version control system. With minimal effort, any developer can clone a public repository, make changes, and then commit those changes back to the repository. Because the repository is local to each developer's machine, developers are able to commit as often as they want [5]. They may not be able to push patches back to a principle branch without approval, but at the very least, they can get the benefit of using a version control system without needing approval by another person.

The issue of productivity with a lack of connectivity is also solved by DVCSs. Having a local repository with a complete version history means that a developer can still view past versions of a project even when not connected to the Internet [13].

An interesting difference between CVCSs and distributed version control systems *(DVCS)* is the granularity in what a developer is able to commit to the repository. When it comes to CVCSs, the atomic unit of commitment is an individual file [17]. While at first it may not seem important, this fact has an interesting effect on the commit habits of developers. According to a research survey, the average size of a commit in terms of lines of code is larger for CVCSs [17]. This can largely be attributed to the fact that the atomic unit of commitment for DVCSs is individual lines instead of whole files.

As a result of the atomic commit unit being an individual line, commits made in DCVSs tend to have fewer lines of code than commits made in CVCSs[17]. One thing not touched on previously, however is that because individual lines can be

committed, changes to an individual file can be recorded over multiple commits. This would be useful if a developer wanted to try to group commit contents based on the intent of the commit, such as fixing a bug, implementing a new feature, tweaking the efficiency of a function and so on. In fact, the research survey conducted by Brindescu *et al.* [17] shows that, at least among the surveyed developers, a DVCS is used to do exactly that: organize changes to be committed together based on their intent.

Another interesting difference between CVCSs and DVCSs is their perceived 'ease of use'. According to the research survey conducted by Brindescu *et al.* [17] although CVCSs are waning in popularity among the development teams surveyed, CVCS was more popular the larger the team was. Additionally, teams using CVCSs overwhelmingly indicated that their continued use was due to perceived 'ease of use' and to familiarity with the tools. Perhaps this should not be surprising considering that many of the most popular version control systems up until roughly 2005, such as CVS and Subversion, were CVCSs.

Based on the characteristics mentioned previously, there are certain situations where a DVCS seems like the more attractive solution. If developers of a project are largely separated by geography, then using a DVCS is potentially a smart choice. If developers anticipate often having long stretches with no sort of Internet or intranet connection then using a DVCS is probably a smart choice. These are by no means the only considerations that should take place, just like how section 2.2 did not touch on all the things that should be considered before going with a CVCS. However, at

the very least the scenarios mentioned indicate when a DVCS should be a strong candidate when choosing a version control system.

Chapter 3

Formiga

3.1 Introduction

Before discussing how I changed Formiga, I will discuss what Formiga does as it was originally developed by Hardt [1].Formiga is a tool for the Ant build system that aids in build maintenance and dependency discovery. It is implemented as an Eclipse plugin and the work in this thesis builds upon that plugin. This chapter will go over the main features of Formiga:

- assisting developers with build maintenance due to external changes
- assiting in the identification of build dependencies in software projects
- assisting in build maintenance due to internal changes

3.2 Build Maintenance Due to External Changes

Often during the lifetime of a software project files will be added, removed, renamed, and moved. When these events occur, developers may need to make corresponding changes to the build system. Formiga, which is implemented as an Eclipse plugin, is capable of recognizing when these events occur within a project. Additionally, it can detect when these actions are performed outside of Eclipse and directly on the file system. When these operations are detected by Formiga, it can update appropriate tasks and properties within the build system. It can do it both automatically or by it asking the user if the build system should be updated. The manner in which it updates the build system in the face of file refactoring depends on the operation is performed (add, remove, rename, move), the build tasks referring to the refactored file, and whether the related references to that file are considered direct or indirect. Hardt considers a direct reference to be a build system reference that can only be resolved to a single file or directory [1]. An indirect reference is a build system reference that might refer to multiple directories through the use of wildcard characters in a task or property.

When a file is added, Formiga reports to the user which targets and tasks will be "directly" affected by the new file. A task is considered to be "directly" affected by a file if the task acts on all files in the directory the file was added to or if the task has an indirect reference to that file.

When a file is renamed or moved, Formiga's behavior differs depending on

whether a task has a direct or indirect reference to the file. If a task directly references a renamed or moved file, the reference is updated because it would otherwise be invalid. If the reference is indirect and the reference still refers to the modified file, then no change happens and the user is not notified of anything. If the indirect reference no longer refers to the modified file, then either a new reference is added to the existing reference or a new reference is included as a nested task.

When the file is removed, Formiga's behavior again differs based on whether a task has a direct or indirect reference to the file. If the task has a direct reference, then the reference is removed. If the removed file was the only file referenced by the affected task, then the task is also removed. If the file is indirectly referenced by a task, then nothing happens.

3.3 Build Dependency Identification

One major feature of Formiga is the identification of "build dependencies", which are relationships between two files that are specified as part of a task in a build system. Formiga starts its dependency analysis by first identifying all the targets in a build file that no other targets depend on. Executing each of these targets would cause the Ant build system to eventually execute all targets, producing a set of what Hardt calls "target chains" [1]. By identifying build dependencies while executing all target chains, Formiga ensures that it finds dependencies for all possible chains within the build.

Formiga uses a modified version of Ant in order to discover a software project's build dependencies. Formiga's Ant implementation allows it to execute all tasks in a build system in nearly the same way as the regular Ant implementation. However, instead of having tasks execute their effect on the user's file system, Formiga's Ant implementation works on a virtual filesystem in memory that Hardt refers to as the "filespace". The filespace maps filesystem locations to all the files that are found in that location. Because Formiga works on a virtual filesystem, it contains models that represent the files within that filesystem. Formiga has models for the following file types:

- Source files
- Class files
- External libraries
- Build files
- Deliverables
- All other files

The class file model contains a set of class files and external libraries. These sets represent the files that are directly used to generate the containing file. The deliverable model contains a set for each of the files identified in the list above. As with the class file, each of the sets in the deliverable model represent the files that

are used to generate the deliverable. The files in these sets are the dependencies identified by Formiga when it executes the Ant build process. When executing the Ant build, Formiga gives each target chain its own filespace. However, Formiga also can identify when target chains have overlapping target subsets and reuses build dependencies that have already been identified.

Formiga may attempt to identify and record build dependencies when either a build file is modified and saved or when a file is added, removed, moved, or renamed. If the build file is modified by a user, Formiga makes no attempt to determine the differences in dependencies between two build system versions and instead recalculates all dependencies. If a file is added, removed, moved, or renamed then Formiga will wait to determine if dependencies need to be reprocessed until it checks for build maintenance updates. As in the previous case, if Formiga needs to recalculate dependencies then it will recalculate all of the dependencies for a project. Dependencies are recorded in an embedded Apache Derby database by using Hibernate. Apache Derby is a Java-based SQL database [18]. Hibernate is a popular Java object relational mapping framework [19]. When Formiga comes across conditionally set properties, it executes the target chain both with and without the property defined in order to ensure that Formiga investigates all possible configurations.

Formiga displays build dependencies as a directed graph. Within the graph nodes represent files and edges represent a dependency between them. It can display both forward and backwards dependencies. File A is a forward dependency of file B if A can only be produced if B is present. In this scenario, B is also considered a

backward dependency of A. When calculating dependencies, Formiga records the line number in a build file that is responsible for constructing a given dependency. When a user clicks on an edge between two nodes on the graph, Formiga uses this information to open the build file at the appropriate line within Eclipse. Users can access the graph through a 'Formiga' option that has been added to the context menu generated when a user right-clicks a file or directory within the Eclipse file explorer.

3.4 Build Maintenance Due to Internal Changes

The final major feature of Formiga is that it provides build system refactoring options similar to what a developer would find in a modern IDE. Formiga provides these refactoring operations because they can be error prone if performed manually and may require a large number of updates.

The first refactoring operation provided by Formiga is target removal. A user can highlight the name of the target at its declaration and select "Remove Target" from its context menu. Formiga will then remove the target and all references to it within the build file. The second refactoring operation is target renaming. The renaming operation is triggered by highlighting the name of the target at the target's declaration and selecting "Rename Target" from the context menu. This causes Formiga to rename all references to the target throughout the build file. The final two refactoring operations are "Property Renaming" and "Property Removal".

These operations are triggered in the same way as target removal and target renaming, however the user has to highlight the name of a property instead of a task. One behavior that is also similar between all four refactoring operations is that upon completion Formiga will report the number of updated property or target references to the user.

Chapter 4

Implementation

4.1 Introduction

FormigaV2 is the name of the updated version of Formiga presented along with this thesis. FormigaV2 differentiates itself from the original Formiga by tracking how dependency history changes over time within a project. It does this with a mock version control system that is made up of two databases:

- *Uncommitted File Database* - The database that stores modified but uncommitted files and build dependencies
- *Committed File Database* - The database that stores committed files and build dependencies along with an identifier that shows when changes were committed

Each database has a table for the file models identified in Chapter 3 as well as a table for all build dependencies.

This chapter addresses the design choices made in this thesis and the rationale behind those choices.

4.2 DVCS or CVCS

The first question coming into this thesis was whether work would be done against a mock version control system of the distributed or centralized paradigm. Ultimately centralized version control seemed to be the best option for a few reasons:

- Having one central authority removes the need to reconcile databases between multiple developers. Because the mock version control system also assumes there is only one authoritative 'committed file' database, transitioning from the mock system to a production-quality system can be relatively painless.

- There isn't as much need to worry about how the state of the committed file dependency database has to be transmitted to developers. The committed file dependency database can just be stored on the repository server and accessed via a connection to that server. Even though the dependency history cannot be retrieved without a connection to the repository server, this behavior is consistent with how most CVCSs function.

- Merging the committed file dependency databases can be ignored in the initial version of the FormigaV2 enhancements. This is not to imply that branching and merging is not important in a CVCS. However, it would be impossible to

present a project that does not solve the issue of merging because the committed file dependency database has to be available to all developers offline. Requiring developers using a DVCS to have an internet connection in order to use all features of FormigaV2 would go against the spirit of DVCSs. By choosing to mock a CVCS we can make the simplifying assumption that all developers are working with one committed file dependency database and not their own potentially different version of that database.

The choice to create a mock CVCS was made mostly out of convenience. Using a CVCS allowed some simplifying assumptions to be made that shifted focus onto the question of how to capture and commit dependencies instead of how to merge the dependencies.

4.3 Database Interaction

One of the technologies used by Formiga is the Hibernate Object/Relational Mapping framework[19]. Hibernate is a data persistence framework that maps Java objects to database tables. Ultimately, I chose not to use Hibernate in FormigaV2 and instead used JDBC which is a database API for the Java programming language. In order to explain why I chose to use JDBC over Hibernate, I will briefly discuss the benefits of both and the reasoning behind my decision.

There are a number of features of Hibernate that make it a fairly popular choice among Java developers. It has multiple options for table initialization and data

fetching that make it relatively simple to tune database performance. Hibernate is built to be scalable which means it fits well into projects of any size. Wide use among Java developers has lead to an abundance of offline and online learning material and help information, which can make it significantly easier to learn how to use Hibernate. Also, the Hibernate Java API abstracts away the choice of database (MySQL, Derby, etc.) from the developer. This abstraction allows the developer to write code that can be reused regardless of what database Hibernate is connecting to.

There are multiple benefits to using JDBC as well in a project. Developers use a structured query language *(SQL)* to interact with a database when using JDBC. Because SQL is a standard topic in university curriculums, many developers have some level of experience with it. JDBC is also quick to set up because its API is included in the Java SDK. Thus, no extra JAR files need to be included in the build path when using JDBC. Additionally, developers do not need to set up configuration files to interact with their database through JDBC. A developer simply uses the database's URI to create a connection and then uses the JDBC API to execute SQL queries. The result is a set of database table rows whose columns can be accessed either by their name or by their position.

Ultimately, the decision to use JDBC over Hibernate stemmed from the desire to do as little to the original Formiga plugin as possible. Because Hibernate directly maps all the fields of a class into columns in a table, storing new information would mean tampering with the original classes. Modification was avoided primarily out

of fear that modifying Formiga might accidentally break it. Other design choices were motivated by a desire to only add code to Formiga, except when needed to fix bugs. One such design choice was to use wrappers around certain data classes instead of modifying those classes to hold extra data.

4.4 Mock Version Control System

The mock version control system is primarily represented by two embedded Apache database tables. One database table is referred to as the "uncommitted files database" and keeps track of the modified but uncommitted files. This table emulates how a version control system will tell the developer which files have been changed since the last time files were committed. The easiest way to keep track of the files within the project is to record each addition, removal, or modification. In addition, by storing the uncommitted files and their dependencies in the database, dependencies only need to be calculated when the relevant files are first modified. The database itself has a table for each type of file identified by Formiga: class file, source file, deliverable, library, and other. Build dependencies also have their own tables based on the type of the two files involved in the dependency. The primary key for each type of file within the database is a combination of file name, file path, and file "version"–a value which is intended to represent a file iteration but currently is unused. Additionally each file is associated with its containing project, but the project is never part of the primary key and, therefore, not required to uniquely identify a

file.

The second database is referred to as the "committed file database". This committed file database has all the same tables as the "uncommitted" table, but with one exception. In order to mimic the "revision" concept from real version control an extra table that records a "commit record" exists within the committed file database. Using this table, each file is uniquely identified by a combination of file name, file path, file version, and "commit record". Additionally, file-to-file dependencies, in addition to being identfied by the two files that comprise the dependency, also have their own associated "commit record".

Eclipse workspace listeners, already used by the original Formiga, are also used to help keep track of the file dependency history. When a file change is detected by the workspace listener, a new row is inserted into the uncommitted file database with one of three values for the update type column:

- *ADD* - The associated file has been added to the project
- *REM* - The associated file has been removed from the project
- *MOD* - The associated file's contents were updated and saved

In general, once a file has been recorded in the uncommitted file database, no further changes are recorded for that file until it is committed. There is one exception to this rule, which happens when a given file exists within the database with the "ADD" update type and FormigaV2 tries to add the same file to the database with a "REM" update type, then the "ADD" row is removed completely and the "REM"

row is never inserted. This exception is intended to mirror the situation in which a developer adds a new file to a project but deletes it before it is ever committed. If a file is added to a version control system and removed before it is committed, the version control history records nothing instead of recording both an add and a remove.

The inverse scenario to the one described above does not display any special behavior. If a file is recorded in the uncommitted file database with a "REM" update type, and FormigaV2 attempts to add a new database entry for the same file but with the "ADD" update type, then the "REM" row is not deleted. If a developer were to remove a file from a project, there is no reason they could not add a completely new and completely different file with the same name to the same location. In this case, it would not be appropriate for the add operation to cancel out the remove operation because adding a file in this scenario does not guarantee it has undone the remove operation.

4.5 Dependency Calculation

Formiga, without any modification, calculates the interfile dependencies for all projects in an Eclipse workspace. The difficulty in determining the interfile dependencies for "uncommitted" files lies mostly in identifying the dependencies for uncommitted files from the objects generated by Formiga's Ant build simulation. Unfortunately, there was originally no way to access the results of simulating Ant's

build process. I modified FormigaV2 to identify the interfile dependencies of uncommitted files over a multi-step process, detailed below:

1. When a change is detected by the Eclipse workspace listeners, Formiga simulates the Ant build process to determine if any file dependencies have changed. During this Ant build simulation, the filespace of each target chain is merged into one universal filespace.

2. FormigaV2 iterates through every file in the filespace, looking for ClassFiles and Deliverables. All of the build dependencies that are recorded within the FormigaV2 databases are ClassFile-to-file dependencies or Deliverable-to-file dependencies, which is why FormigaV2 looks for ClassFiles and Deliverables. When a ClassFile or Deliverable is identified, then FormigaV2 iterates over the list of added, removed, and modified files identified by the Eclipse workspace listener.

3. If the file from the filespace is a Deliverable, then FormigaV2 checks if the relevant file is in one of the file sets for that Deliverable. If the file from the filespace is a class file, then FormigaV2 checks if the relevant file is in one of the two file sets within that model. When a file is in one of these sets within the Deliverable or class file then that file is directly used to generate that deliverable or class file.

4. If the added or removed file is found within one of the "generated files" set for either the deliverable or class file, then the deliverable or class file and the

added or removed file form a valid file-to-file dependency. The two files are then added as a new row to the uncommitted files database based on what the type of the two files are, with the update type corresponding to whether the second file was added or removed.

In this way, when file changes are detected by the Eclipse workspace listener, appropriate file dependencies are identified and recorded in the uncommitted file database. Files and file-to-file dependencies stay in the uncommitted file database until the developer decides to commit the work. When a user decides to commit a set of changed files using the mock version control system, they can access the "Uncommitted Changes" window. As shown in Figure 4.1, the uncommitted changes window shows all uncommitted files for each project in the workspace. The developer specifies the files to be committed by checking the checkbox next to the desired files and selecting the 'Ok' button at the bottom of the window.

When a developer selects 'Ok' in the uncommitted changes window, in addition to moving the selected files to the committed files database, the appropriate file dependencies are also recorded in this database. When moving individual files to the committed file database, if the selected file is not a part of any file-to-file-dependencies or if the only associated dependency is also going to be committed then the file is completely removed from the uncommitted file database.

When individual files are committed, all relevant file-to-file dependencies are also committed as well. If a selected file is either a deliverable or a class file, then

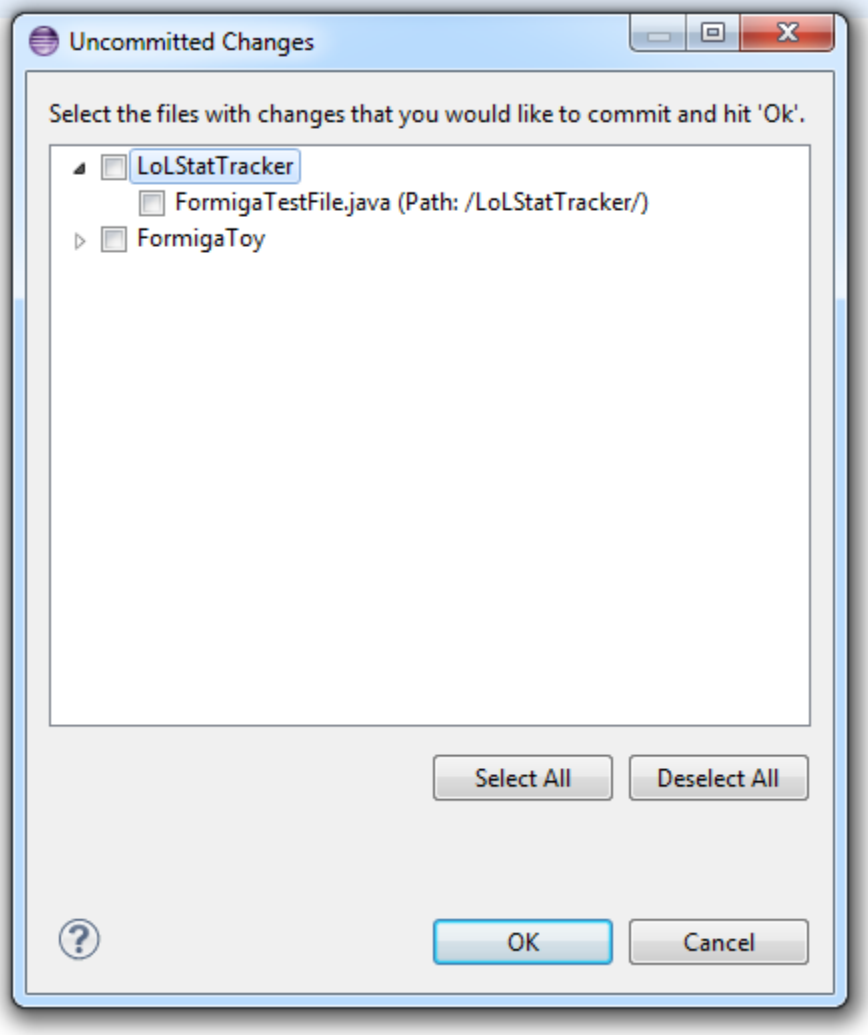

Figure 4.1: Uncommitted Changes Window

some extra calculations are performed to determine whether there are any relevant dependencies to commit. FormigaV2 first creates a commit record to associate with all the files about to be committed. After that, it iterates over all the files that are about to be committed, and searches the uncommitted file database for any dependencies that file might be involved in. This process creates two maps, one from a class file to all the files used to generate it and the other is from a deliverable to all the files used to generate it. The combination of a deliverable or class file with one of the files within the mapped sets is how FormigaV2 represents the build dependencies before they are recorded. Next, all uncommittable dependencies are pruned from these two sets. Dependencies are considered committable in three situations:

1. If both the generator file and the generated file are being committed in the current revision, the dependency can be safely committed.

2. If the generator file is being committed, but the generated file is not being committed, FormigaV2 checks the committed file database to see if there is a record of the generated file being added or modified. If such a record is found, then the dependency is committable.

3. If the generated file is being committed but the generator file is not being committed, FormigaV2 checks the committed file database to see if there is a record of the generator file being added or modified. If such a record is found, then the dependency is committable.

If a dependency does not meet one of these three criteria then it is removed from the file dependency map. In other words, a dependency is considered uncommittable if only one of the two files in the dependency is being committed and the other file has never been committed. Dependencies should not be recorded if both files are not stored in the commmitted files database. Finally, after all the uncommittable dependencies have been removed, all the remaining dependencies are recorded in the committed file database and removed from the uncommitted file database.

4.6 Completeness of Solution

When talking about the completeness of FormigaV2, it seems appropriate to ask if it captures file dependencies in all the same ways as Formiga. In its original implementation, Formiga recognizes that the build dependencies of a software project may change in two distinct situations:

1. A file is added, removed, renamed, or moved

2. The build system is manually modified by a developer

It is fair to say that FormigaV2 captures all the dependencies generated by the first situation. Formiga treats renamed files and moved files as a sequence of additions and removals, where either the name or the path of the file respectively has changed. FormigaV2 also treats a rename or move operation as a sequence of addition and removal operations. Then, to verify that all four operations in FormigaV2 capture

the same dependencies as Formiga, it is sufficient to verify that the addition and removal operations work the same. I have verified that both operations work as expected by comparing the Derby database used by Formiga to the uncommitted file database used by FormigaV2. When a file is added to a software project, I verified that FormigaV2 captures all the correct dependencies by confirming that all dependencies for the added file in embedded Formiga database are also found in the uncommitted file database. I verified that the remove operation captures the appropriate dependencies in FormigaV2 in a similar manner. When a file is removed from the project, I determined that all the dependencies listed as "removed" in the uncommitted files database are not present in Formiga's embedded database.

Unfortunately, FormigaV2 does not capture new dependencies generated when the build system is manually modified. The reason this situation is not covered is because there is no intuitive way to capture new dependencies identified by Formiga after the build system is manually modified. When dependencies are identified by Formiga in this situation, instead of determining the differences and committing only new dependencies Formiga clears out the entire database and commits the entire filespace. In order to capture these dependencies FormigaV2 would have to do one of two things. FormigaV2 could compare the dependencies between the in-memory filespace and the Hibernate database, but the computational cost of this method is why Hardt avoided it in the first place. The second option would be to modify Formiga's implementation of Ant to check each identified dependency for existence inside the Hibernate database. If the dependency is not already recorded

then it must be new. The second option would have been my choice to handle this dependency generation scenario. However, as mentioned in Section 4.3, I tried to avoid modifying Formiga in order to avoid potentially breaking it and so I chose to capture dependencies generated by the first dependency generation scenario instead.

The only remaining question is whether or not the interface is a sufficient replacement for a production-quality version control system. Referring back to Figure 4.1, note first that all files are listed beneath their project. Additionally, each file specifies its path so that there is no confusion between multiple files that may have the same name within a project. Added, removed, and modified files are represented in a manner similar to a production-quality version control system. However, files that are moved or renamed could be improved upon. Because a renamed or moved file is treated as a sequence of removal followed by addition, when one of these two operations is performed two items show up within the uncommitted changes window. To commit a file that is modified by either the rename or or move operations a user has to choose to commit both the removal of the old file and the addition of the new file. This is not the same behavior as a production-quality version control system, so it would not be very intuitive to the average developer.

4.7 Issues with Implementation

The most troublesome issue I encountered while developing FormigaV2 was dealing with Hibernate. As mentioned in Section 4.3, Formiga uses Hibernate to interact

with an embedded database. When I began working on FormigaV2 I contacted Hardt and he gave me the JFreeChart project (the Java project he used in his usability studies) [1]. When calculating the dependencies of the JFreeChart project, Formiga was unable to record the filespace. The issue was that Hibernate was throwing a *NonUniqueObjectException* in the middle of recording the filespace. Investigation showed that this exception is thrown by Hibernate if a program attempts to save multiple objects that represent the same component (based on the component's overwritten Java equality methods). Based on my own conversations with Hardt[20], he had struggled with similar issues when implementing Hibernate in Formiga. After multiple weeks of correspondence between myself and Hardt, we were still unable to solve my NonUniqueObjectException. Because Hibernate does not throw a NonUniqueObjectException when it calculates the dependencies of my toy project, I decided that my time was better spent working on FormigaV2 rather than resolving my issue with the original Formiga implementation.

Chapter 5

Future Work

5.1 Production-Quality Version Control Integration

The initial effort to integrate Formiga with a version control system was done using a mock version control system. The next logical step would be to integrate Formiga with a production-quality version control system. It is likely that the easiest way to go about this would be to choose a CVCS such as Subversion for integration. As noted in chapter 4, the mock version control system in this thesis was designed to be similar to a traditional CVCS. Integrating with a production-quality version control system would allow Formiga to use the revision identifier used by the version control system. Additionally, it might be possible for Formiga to only track file-to-file dependencies and stop tracking individual files because a version control system

already keeps track of file history. An integrated Formiga might require the least changes if it is integrated with a version control system that is also integrated into Eclipse, such as Eclipse Subversive[21]. Ultimately, Formiga will be most useful to developers if it is integrated with a real version control system instead of a mock system.

5.2 Distributed Version Control System

After integrating Formiga with a CVCS, it could be worthwhile to also integrate it with a DVCS. As noted in chapter 2, within the past decade DVCSs have begun to gain popularity with developers at least in part due to having a complete repository–history included–available locally. Having complete version information available locally means that in order to integrate Formiga with a DVCS it would also have to be able to transmit partial information about the change of the committed file dependencies. As mentioned in section 5.1, Formiga would likely be most useful if integrated with a DVCS that is also integrated into Eclipse, such as EGit[22].

5.3 Visualization

One big component of Formiga that is ignored by this thesis is the task of visualizing dependency history. One important feature of Formiga is the ability to visualize dependencies within a build system and make it easier for a developer to understand

the structure of that build system. The first issue that would need to be resolved is building routines that can retrieve all of the file dependencies from a given point in a repository's history. When all the dependencies for a given repository version have been gathered then they can be used by the routines responsible for displaying dependencies. A developer might also be interested to see how the dependencies of one file have changed over time. The biggest hurdle showing this for a given file is finding an intuitive representation for the dependencies. It could be difficult to cleanly represent the dependency between two files if, for example, one of the two files has been added and removed to and from a project multiple times.

Chapter 6

Conclusion

Formiga helps developers to better understand their Ant build system by providing automated build maintenance tools and visualization capability, but it does not necessarily help a developer understand how the build system has changed over time. This thesis begins to fill that void by laying the groundwork for integration with a production-quality version control system.

Formiga is modified to use mock CVCS in order to determine how dependencies should be represented over time and how user interaction with version control would record those dependencies. Using Eclipse workspace listeners, the Formiga plugin can track when files in a project are added, removed, or modified and record the interfile dependencies specified by the Ant build system. Though developers are not yet able to interact with the dependency history in a meaningful way, the dependencies are available for future developers to use if they choose to expand the Formiga Eclipse plugin.

Bibliography

[1] R. Hardt, *Ant Build Maintenance With Formiga*. PhD thesis, University of Wisconsin-Milwaukee, May 2014.

[2] B. Adams, "Co-evolution of source code and the build system," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 461–464, Sept. 2009.

[3] "Cvs - concurrent versions system." `http://www.nongnu.org/cvs/`, April 2015.

[4] P. Collins-Sussman and B. Fitzpatrick, "Version control with subversion(for subversion 1.7)."

[5] B. de Alwis and J. Sillito, "Why are software projects moving from centralized to decentralized version control systems?," in *Cooperative and Human Aspects on Software Engineering, 2009. CHASE '09. ICSE Workshop on*, pp. 36–39, May 2009.

[6] "git." `http://git-scm.com/`, April 2015.

[7] "Mercurial." `http://mercurial.selenic.com/`, April 2015.

[8] W. Swierstra and A. Löh, "The semantics of version control," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, (New York, NY, USA), pp. 43–54, ACM, 2014.

[9] M. Rochkind, "The source code control system," *Software Engineering, IEEE Transactions on*, vol. SE-1, pp. 364–370, Dec 1975.

[10] W. F. Tichy, "Rcsa system for version control," *Department of Computer Sciences Purdue University*, 1995.

[11] M. Manson. private communication.

[12] N. Bertino, "Modern version control: Creating an efficient development ecosystem," in *Proceedings of the 40th Annual ACM SIGUCCS Conference on User Services*, SIGUCCS '12, (New York, NY, USA), pp. 219–222, ACM, 2012.

[13] D. Spinellis, "Git," *Software, IEEE*, vol. 29, pp. 100–101, May 2012.

[14] "Bazaar." `http://bazaar.canonical.com/en/`, April 2015.

[15] "Bitkeeper." `http://www.bitkeeper.com/`, April 2015.

[16] L. Torvalds, "Re: Kernel scm saga..."

[17] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 322–333, ACM, 2014.

[18] "Apache derby." `https://db.apache.org/derby/`, April 2015.

[19] "Hibernate. everything data. - hibernate." `http://hibernate.org`, March 2015.

[20] R. Hardt. personal communication.

[21] "Eclipse subversive: Subversion (svn) team provider." `http://eclipse.org/subversive/`, March 2015.

[22] "Egit." `http://eclipse.org/egit/`, March 2015.