ANT BUILD MAINTENANCE WITH FORMIGA

by

Ryan Hardt

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

Doctor of Philosophy
in Engineering

at
The University of Wisconsin-Milwaukee
May 2014

ABSTRACT
ANT BUILD MAINTENANCE WITH FORMIGA

by

Ryan Hardt

The University of Wisconsin-Milwaukee, 2014
Under the Supervision of Professor Ethan V. Munson

A build system produces a set of deliverables from a software project's source code and resources. "Build maintenance" refers to the changes made to the build system as a software project evolves over time. It has been shown to impose a significant overhead on overall development costs, in part because changes to source code often require parallel changes in the build system. However, little tool support exists to assist developers with build maintenance, particularly for those changes that must accompany changes to the source code. Formiga is a build maintenance and dependency discovery tool for the Ant build system. Formiga's primary uses are to automate build changes as the source code is updated, to identify the build dependencies within a software project, and to assist with build refactoring. Formiga is implemented as an IDE plugin, which allows it to recognize when project resources are updated and automatically update the build system accordingly. This implementation also allows it to leverage existing metaphors used by developers to maintain source code, thus making it easier to use. A controlled experiment was conducted to assess Formiga's ability to assist developers with build maintenance. Formiga was shown to signficantly reduce the time required to perform build maintenance while increasing the correctness with which it can be performed.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Build systems are a necessary component of software engineering, responsible for producing a set of deliverables from a project's source code and resources. These deliverables include artifacts like executables, reusable libraries, and archives of source code or documentation. Many software projects (particularly large projects) have a build system that is used to produce multiple deliverables, thus increasing the size and complexity of the build system. Additionally, configurations may be used to control which deliverables are produced, or for which system a deliverable is to be built. These configurations often dictate which files are involved in the build process or which build targets are executed. Configuration values may be determined at runtime (based on the environment used to build the deliverable, for instance) or by specifying a pre-determined set of values. This results in different build execution paths for each of the configuration sets, thus further complicating the build.

Various stakeholders in the software development process require information about the dependencies between artifacts in a software project [1]. Much of this information is stored in the build system. Software developers require knowledge of the build system so that they can make updates to it as project resources are modified. Software maintainers require an understanding of the build system to ensure that any maintenance tasks are reflected in the appropriate deliverables. Project managers and those responsible for application deployment may need to know which deliverables or applications will be affected by changes to one or more files in a software project. All of these tasks require knowledge of the dependencies represented in a build system.

Due to the size and complexity of many build files, modifications can be time consuming, challenging, and error-prone. When project files are created, removed, or updated, the build system may require or benefit from updating. Tasks may reference the modified files directly, directly reference the directories containing those files, use patterns to reference those files or directories, or reference a property whose value refers to those files. This makes it more complicated to identify where those files are being used in the build system, which makes the maintenance process more challenging.

Like source code files, build files are also subject to refactoring. For example, deliverables could be added, removed, or have their contents changed. This requires adding, removing, or modifying build targets, which some developers may be hesitant to do in fear of "breaking the build". Because some development teams

disperse build updates among the developers working on the project rather than using a dedicated team of developers for build development [2, 3], this hesitance may be a significant impediment to appropriate build updates. Incomplete build refactorings may indeed break the build or result in unused ("dead") build code, which may be subject to further refactoring that is, in-fact, unnecessary.

"Build maintenance" refers to changes that are made to a software project's build system as it evolves. Prior research has shown that the build system needs to evolve in parallel with the source code, and that it grows in size and complexity as the source code does [4, 5, 6]. Build system maintenance alone imposes a 12%-36% overhead on the development of a software project [7]. Additionally, up to 27% of work items involving production source code changes require accompanying build maintenance [2, 3].

A project's build system may require updates in response to both internal and external changes. More precisely, build maintenance may occur for two reasons:

1. refactored source code requires build changes (external), or

2. the build system itself needs refactoring or fine-tuning (internal).

"Build maintenance recommendation" was identified as a potential means to reduce the overhead imposed by build maintenance caused by external changes [3]. Because source and build code co-evolve, it is easy to imagine situations in which a developer is unaware that changes made to the source code require build code changes. Build maintenance recommendation is intended to inform developers of

these source code changes that require build maintenance. Moving a file from one directory to another or simply renaming a file are two such operations. Without updating the build system, these operations may cause unexpected build results or break the build process altogether. If developers are not notified immediately after making such changes to the source files, it may be difficult to determine the cause of an erroneous build process, particularly if the build system is complex or contains multiple configurations. While recent research has greatly improved the state of build maintenance tools [8, 9], support for automatic build maintenance due to external changes is lacking. This is the primary motivation for our build maintenance tool, *Formiga*.

To make Formiga a useful build maintenance tool, it must also address build maintenance caused by internal changes, which it does by automating various build refactoring tasks, such as target renaming, target removal, variable renaming, and variable removal. These tasks may be error-prone if performed manually due to multiple build files in a project, variable assignments made at runtime, or general unfamiliarity with the build system. Furthermore, Adams *et al.* [8] identified the following five requirements that a reasonable build maintenance system should provide: visualization, querying, filtering, refactoring, and validation. Formiga aims to meet these requirements by facilitating dependency discovery and assistance with both types of build maintenance for the Ant build system.

The remainder of this dissertation is organized as follows. Chapter 2 presents relevant background information. Chapter 3 discusses related research that focuses

on build system evolution, overhead, and maintenance. Chapter 4 presents the implementation of our build maintenance tool "Formiga". Chapter 5 describes a controlled experiment used to assess Formiga's ability to assist developers with build maintenance. Formiga's constraints are presented in Chapter 6, and its contributions are outlined in Chapter 7. Chapter 8 outlines future research involving Formiga, and Chapter 9 concludes this paper.

Chapter 2

Background

This section describes terminology and concepts that will be used throughout this paper. Section 2.1 describes build tools, including Ant, Make, Maven, Gradle, and those implemented within software configuration management (SCM) systems. Section 2.2 discusses ways in which a build system can be analyzed to discover information about the deliverables it produces.

2.1 Build tools

Build tools are capable of executing and automating a variety of tasks, including:

- Compiling source code to create executables
- Packaging files into binary files for distribution
- Deploying executables or binary files for release

- Running software tests
- Creating documentation files

Proper use and understanding of build tools and processes is important, particularly for large software projects [10]. Two widely used build tools are Make [11] and Ant [12]. Both of these build tools allow one to specify how to construct derived artifacts using a set of tools and project files. We chose Ant as the build tool analyzed by Formiga due to familiarity with Ant as well as its benefits described by Serrano and Ciordia [13]. Two more recently introduced build tools are Maven [14] and Gradle [15].

2.1.1 Make

Make [11] performs tasks based on the contents of a *makefile*. Makefiles are written in a "tabbed text" format, where tab characters and other whitespace are used to indicate relationships between the components present in the makefile. Makefile components consists of *rules*, *targets*, *dependencies*, and *commands*. A *rule* is a named collection of commands, identified by a target. A *target* is often the name of a file to produce, or simply a name associated with a collection of commands. A target may have an associated list of dependencies. A *dependency* is a file that is used as an input to a command in the rule for the associated target. A makefile may reference components specified in another makefile.

Make is run by executing a specified target. When a target is executed in Make, if

any of the rule's dependencies are missing or have a more recent timestamp than the target, then (and only then) will make first update those dependencies by executing the rules used to produce those dependencies. After all of its dependencies have been updated, a rule's target is built by executing its associated commands. This methodolgy avoids unnecessary exeuction when a target's dependencies have not changed.

Make can build projects written in any language. It was designed to run on a Unix operating system but has been ported by various third parties to run on other operating systems as well. All of the commands present within a makefile refer to arbitrary Unix programs, some of which may need to be obtained separately from Make. Because the programs associated with these commands may not be packaged with Make, documentation for them may be difficult to find.

2.1.2 Ant

Ant [12] performs tasks based on the contents of a build file written in XML, typically named "*build.xml*". This build file primarily consists of *targets*, *properties*, and *tasks*. Each build file consists of a single *project*, which is a named collection of targets and specifies a base directory by which all relative paths will be related. A *target* is a named collection of executable tasks that may depend on other targets. A *task* refers to an executable piece of code. A task may contain other tasks. A *property* is an immutable name-value pair that may be defined at the project level or at the

target level. An Ant build file may reference components written in another Ant build file.

When invoking Ant, one specifies a target to execute. When a target is executed, all dependencies of the target must be executed first, regardless of the presence or timestamps of any dependent files. A specified target or target dependency is always executed in Ant unless the target contains either an *if* or *unless* attribute whose value indicates that the target does not need be executed. When executed, however, some Ant tasks can internally determine that their work is already done. This determination varies by task.

Ant can build projects written in any language but is most often used to build software projects written in Java. It can be run on any operating system with a Java virtual machine (JVM) installed. Most of the tools executed during an Ant build are written in Java and packaged with the Ant distribution. Their behaviors and invocation syntax are well documented. Additional Ant tasks can be written in Java and referenced in a build.xml file.

2.1.3 Maven

Maven [14] uses a "pom.xml" configuration file to define a single artifact to build for a software project. Rather than identifying the procedures used to build this artifact (as done in a makefile or build.xml file), a pom.xml file consists of declarations that identify the artifact to produce, where its contents are located, and what

external dependencies it has. To facilitate this, Maven imposes restrictions on the organization of a software project as well as the tools that it uses to produce that project's artifacts. A pom.xml file can refer to other pom.xml files to allow multiple artifacts to be produced.

Maven provides a uniform build system as its builds are based on the concept of a "build lifecycle". When executing a build, the desired lifecycle phase is specified. These phases are predefined and depend on other phases. Most projects can be built by issuing the "install" command (or phase), which builds the desired artifact and installs it at the specified repository. Other commands can be used to separately compile, test, or package the project's source code. Maven can also deploy an executable artifact produced for a software project. Because an artifact's external dependencies are specified with their repository locations, Maven can automatically obtain an artifact's external dependencies when that artifact is built.

Maven is implemented in Java and only builds Java software projects. Plugins can be written to extend the functionality of Maven. If multiple artifacts are desired for single Java project, multiple pom.xml files should be used. The contents of the pom.xml files for complex projects can become larger than their equivalent implementations in Ant. While Maven provides consistency across projects, the constraints it imposes are not suitable for all projects.

2.1.4 Gradle

Gradle's [15] build scripts are written in a Groovy-based domain-specific language (DSL). Groovy [16] is a dynamic, object-oriented programming language. Gradle build files are typically named "build.gradle". A Gradle build is made up of one or more "projects". Projects can be used to build a deliverable or perform a series of operations. Projects can reference other projects. Each project is made up of one or more "tasks". A task is an atomic unit of work. Gradle tasks can depend on other tasks. Tasks can be created dynamically or modified at runtime. Gradle has properties that can be referenced and methods that can be called throughout a build. In addition to executing its own task implementations, Gradle can also execute Ant tasks and import Ant build files.

A Gradle build is often executed by issuing the "gradle" command with the name of a task to execute. Much like Ant, Gradle has a configuration phase and an execution phase. During the configuration phase, a directed-acyclic graph is produced that identifies the tasks to execute. Gradle provides hooks into this graph, allowing developers to modify it before the execution phase occurs. Gradle supports incremental builds, allowing tasks to indicate whether or not their work has already been done during a build execution.

While Gradle can be used to build projects written in any language, its main focus is to build Java projects. It aims to "[provide] the power and flexibility of Ant with the dependency management and conventions of Maven". Like Maven, Gradle

also allows a project to specify the locations of its external dependencies, which can be automatically downloaded during a build. Gradle can also be used to publish artifacts to a repository.

2.1.5 SCM systems

Software configuration management (SCM) systems are responsible for storing and managing collections of files that constitute various software projects. According to Dart [17], SCM systems should include the ability to:

- represent relationships between components
- build binary files from versioned source files
- describe the impacts of a change and provide control over those changes

While the full scope of SCM system functionalities is much larger, this subset of functionalities is related to build tools. As a result, many SCM systems include custom build systems, many of which are variants of Make [18, 19, 20, 21, 22, 23, 24, 25]. Due to their scope, SCM systems are often monolithic in nature with significant learning curves.

2.2 Build analysis

To analyze the effects of a build, a model may be generated of the build system either statically or dynamically. This section discusses both approaches to build

analysis.

2.2.1 Static build analysis

A *static analysis* of the build system evaluates it without actually executing a build. Only the contents of the build files themselves are analyzed. This requires knowledge of the syntax and semantics of the build system as well as information about the tools that are referenced by the build files.

Static analysis has the advantage of having access to build-related data for all deliverables and configurations, not just data for the system on which the analysis occurs. This data can be difficult to obtain, however, partially due to the dynamic behavior of tools that may be executed during the build process.

2.2.2 Dynamic build analysis

A *dynamic analysis* of the build system evaluates it by executing and examining a run of the build. This form of analysis may either observe the build and the environment in which it is executed or evaluate artifacts produced by the build during its execution. Some build tools can produce descriptions of the operations that they perform during an execution, which may be read by a dynamic analysis tool.

Dynamic analysis is likely an easier approach to obtaining a build model than using static analysis, but its data is typically relevant only for a single deliverable

using a particular configuration. If data regarding multiple deliverables is desired, the build may need to be executed multiple times. Additionally, data for multiple configurations may require executing the build multiple times or in multiple environments.

Chapter 3

Related Work

This chapter addresses related research and development tools. Section 3.1 presents research on the development and evolution of build systems. Section 3.2 focuses on SCM systems with custom build systems. Section 3.3 presents research on tools that are focused on build maintenance. Section 3.4 discusses existing commercial tools to assist in the development of Ant build systems.

3.1 Build development and evolution

This section presents research on general build practices and development. It includes a discussion on the importance of build systems, research on how the build system evolves over time along with the software it is intended to manage, and data regarding the costs associated with maintaining a build system.

3.1.1 Build practices

Spinellis [10] studied build practices and stresses the growing importance of build systems due to larger code bodies and sophisticated tool chains. He emphasizes the importance of using build systems to automate, optimize, and polish the build processes.

The author states that automating all build tasks is software building's golden rule. He specifies three purposes served by automation: documenting the processes, speeding up the corresponding tasks, and eliminating mistakes and forgotten steps. Additionally, he recommends optimizing the build, which involves correctly handling the dependencies so that unnecessary steps are not executed. Spinellis states that makefiles and Ant build files are also source code and should be treated as such by storing them in version control repositories and refactoring them when appropriate.

Spinellis warns against using IDE specific build tools (like those addressed in 3.2), as the build specifications become dependent on the IDE and platform they run on. He addresses strengths and weaknesses of various build tools, including Make and Ant, and states that debugging is more difficult in Ant because analyzing tasks involves adding print statements or examining Java source code.

3.1.2 Build evolution

Adams *et al.* presented a case-study of the evolution of the Linux kernel build system [4], which is implemented in Make. They analyzed its growth by measuring

the number of source lines of code (SLOC) as well as the number of targets and dependencies (both explicit and implicit) in its makefiles. This growth was measured for all major releases of the Linux kernel build system from 1991 through 2007.

Their data was obtained using a reverse engineering framework for build systems, *MAKAO*, which will be discussed later in 3.3.1. For each release, they used their tool to generate the build dependency graph of the default build configuration. Using these graphs, they measured metrics like the number of nodes and edges in the graph to determine the number of targets and dependencies in the build. They also obtained additional information from other resources like the project's documentation and mailing list.

Their study showed that the build system of the Linux kernel evolved, that it grew in complexity as it evolved, and that it required considerable maintenance effort in order to deal with the growing complexity. They found that the size of the Linux kernel build system grows exponentially. Their findings also suggested that not only do build systems evolve, but that they co-evolve with the project's source code. Adams later provided four hypothesis for the co-evolution of source code and the build system in [5].

McIntosh *et al.* [6] showed that not only do Ant build systems evolve over time, but they also need to react in an agile manner to changes in the source code. Their study consisted of analyzing build system specifications from a static perspective, where source code software metrics were applied to Ant files, and from a dynamic perspective, where output logs from a representative sample of build runs were

analyzed.

McIntosh *et al.* set out to address the following two research questions: (1) Do the static size and complexity of source code and build system evolve similarly? and (2) Does the perceived build-time complexity evolve? Build-time complexity is a measure of the perceived complexity observed by the build system user. It measures how much build code is routinely exercised and how long a typical build takes.

Their research analyzed official releases of four open source projects ranging in size from small to large. The static metrics used were static build lines of code (SBLOC), build target/task/file count, and Halstead complexity. The Halstead complexity metrics were adapted from source code to build systems. They measure (1) how much information a reader has to absorb in order to understand a program's meaning, (2) how much mental effort a reader must expend to create a program or understand its meaning, and (3) how much mental effort would be required to recreate a program. McIntosh *et al.* believe that their use of a relatively objective measure of build system complexity (the modified Halstead metric) is novel. The dynamic metrics used were dynamic build lines of code (DBLOC), length and depth of build graph, and target coverage percentage. DBLOC measures the percentage of code in the build system that is exercised by the default or clean targets. Source code for each software release was measured in source lines of code (SLOC) as well.

McIntosh *et al.* found that build systems follow either linear or exponential evolution patterns in terms of size and complexity, depending upon the corresponding changes in the source code. These patterns are highly correlated with the evolution

of source code, as SBLOC and SLOC were found to be highly correlated. Major changes in the build systems studied were caused by major changes in the corresponding source code. They also found that the perceived build-time complexity does evolve, but no common pattern was found. SBLOC was found to be a good approximation of the complexity of a build system, as the Halstead metrics were highly correlated with the size of the build system. Target coverage was consistent for each project, with larger fluctuations caused by project restructuring or major releases.

3.1.3 Build maintenance overhead

Kumfert and Epperly [7] studied the percentage of resources devoted to build issues instead of core development for various software projects. "Build issues" here refers to "the development, debugging, maintenance and extension of the supporting infrastructure that converts source code into its end-use form". The objects examined included makefiles, various helper scripts, and the tools used to produce and maintain them. They conducted and analyzed a survey that examined the perceived overhead of a build and analyzed the CVS repository for the software project that was the focus of the survey. The perceived "build overhead" is an estimated percentage of time devoted to the build system compared to the total time spent on software development.

Their survey covered 39 responses from 36 people covering at least 28 different

projects. It indicated that the average build overhead was 11.91% and the median build overhead was 10%. The minimum time spent maintaining the build system was reported as 0% and the maximum was 35.71%.

Additionally, Kumfert and Epperly obtained an objective measure of the build overhead by mining data in CVS for a single project. According to the survey results, the build overhead for this project was estimated to be at least 20%. The project consisted of 1,187 files, 409,858 lines, and 7,984 commits. Each file change in CVS was considered as one unit of work. Build related files constituted 27.5% of the overall number of file changes, and build related line changes accounted for 13.7% of the of the overall line count changes. Kumfert and Epperly indicated that these findings were consistent with the perceived overhead for this project reported in the survey.

McIntosh *et al.* [2, 3] followed their study on the evolution of Ant build systems with an empirical study of build maintenance effort. In their evaluation, they mined the version histories of ten software projects (one proprietary and nine open source) of various sizes to measure the overhead of build maintenance on developers. Their analysis focused on (1) how frequently code changes require build changes and (2) the proportion of developers responsible for build maintenance.

Their study revealed that build maintenance yields up to 27% overhead on source code development and a 44% overhead on test development. These percentages can be interpreted as the percentage of "work items" that require an accompanying change to the build system, where a "work item" is an enhancement or bug fix.

They suggest that project managers should account for this in their project plans. They also found that the build system churn rate is comparable to that of the source code, and build changes induce more relative churn on the build system than source code changes induce on the source code. As a result, the build system may be susceptible to defects.

Additionally, their study showed that up to 79% of source code developers and 89% of test code developers are significantly impacted by build maintenance. For the projects examined, build maintenance was performed either by a small team of build experts or dispersed among most developers working on the project.

Hochstein and Jiao [26] performed a case study of two software projects, *FACETS* and *FLASH*, to determine the amount of effort devoted to maintaining their build scripts. They refer to this effort as the "build tax". Their goal was to provide initial estimates on this build tax, to generate a starting point for future studies, and to motivate the development of better build tools.

Their study recorded three metrics: (1) the percentage of total lines of code in the software project belonging to build-related files, (2) the percentage of regression test failures caused by the build system, and (3) the percentage of build-related commits to the repository. The data related to regression test failures was gathered during a one year period for both projects. The data related to repository commits was gathered over a 6.5 year timeframe for the FLASH project and an 11.5 year timeframe for the FACETS project.

Hochstein and Jiao found that build-related code represented 5% of the total

number of lines of code in the FLASH repository and about 6% for FACETS. FACETS failed regression tests were caused by the build system between 11% and 38% of the time and between 13% and 47% of the time for FLASH. The low end of the range indicates the percentage of failures that occurred in all testing environments, and the high end of the range indicates the percentage of failures that occurred in at least one testing environment. The percentage of build-related commits for the FLASH project was between 19% and 37% and between 58% and 65% for FACETS. The low end of the range indicates commits where all files in the commit were build-related, and the high end of the range indicates commits where at least one file in the commit was build-related. Their results suggest that build scripts are modified far more often than one would expect given the small fraction of overall code that they represent.

3.2 SCM systems

This section presents research on various SCM systems that emphasize the importance of maintaining build-related data. While the scope of SCM systems typically extends beyond that of build maintenance, many SCM systems include functionality to assist developers in producing and maintaining software artifacts. These SCM systems often use custom build tools to accomplish this.

3.2.1 DSEE

In 1984, Leblang and Chase [18] described an SCM system with user-defined dependency tracking named *the DOMAIN Software Engineering Environment (DSEE)*. It consists of a *history manager* used to provide complete version histories, a *configuration manager* used to build systems from their components, a *task manager* that relates source code changes to higher-level activities, a *monitor manager* that watches user-defined dependencies and alerts users when such dependencies are triggered, and an *advice manager* that provides templates for redoing common tasks. DSEE works with any language and allows users to use a text editor of their choosing.

The configuration manager of DSEE maintains a system model in which descriptions of the components that comprise an application, the build dependencies for each component, and the build rules applied to these components are maintained. A component's *build dependencies* include any objects that are relevant to the rederivation of that component. The system model is source oriented but does not specify which source versions to use in a build. A *configuration thread* (CT) states which version of each component in the system model should be used for a build. At build time, the CT is used to bind the components in the system model to particular versions. This *bound configuration thread* (BCT) and keywords describing the system built are stored in a database.

According to Leblang and Chase, "Users should be able to define dependencies

on elements such that other users will be informed of those dependencies before modifying the elements, and such that the user defining the dependency will be informed when the elements are modified." DSEE accomplishes this through its monitor manager which allows users to create *monitors* that define the dependencies present for elements on which they are assigned. A monitor is activated when a new version of any target element is created. The users who monitor the dependency are then notified of the change, and any commands associated with that monitor are executed.

3.2.2 Vesta

Heydon *et al.* [23] developed an SCM system named *Vesta* that allows (1) repeatable builds by storing configurations that refer to immutable component sources and build tools, (2) incremental builds by reusing cached versions of previous builds, and (3) consistent builds by using automatically captured build dependencies to indicate whether reuse of cached results of previous builds is possible. Vesta was specifically designed to work with very large software projects. Other goals were to work with standard development tools and to be easy to use.

Heydon *et al.* identified a number of problems with using Make to build software deliverables. One problem identified is that inconsistent results can be produced by Make if incorrect dependency information is specified or if timestamps are recorded incorrectly. Another problem is that some dependencies, such as those involving

environment variables, are inexpressible, and others, such as those on the makefile itself, are too costly to express. Heydon *et al.* also say that Make's use of timestamps is problematic in situations where a system is built from older sources. Make may incorrectly determine that the system is up-to-date when in fact it is not.

Heydon *et al.* also identified problems with ClearCASE [25], an SCM system based on the DSEE system discussed previously. ClearCASE uses its own version of Make that does automatic (but somewhat incomplete) dependency detection by monitoring and recording the files accessed during a build. It also manages derived files for later reuse. Reusing derived files produced by others is referred to within ClearCASE as "winking in", and Heydon *et al.* say it is based on heuristics that can miss opportunities for reuse. Because it is based on Make, Heydon *et al.* say that the build system of ClearCASE suffers the same scalability problems as Make. They identify another problem with ClearCASE by saying that it may produce inconsistent builds due to the fact that its dependency detection is incomplete. Lastly, they state that the overhead introduced by using ClearCASE's build system may be large enough to cause users to use Make instead.

Vesta uses complete, source-based configuration descriptions. Every element contributing to a build is described in the system model. These elements include all environment components, such as tools, libraries, header files, and environment variables. The Vesta builder reads user-written system models and a set of system-supplied models that constitute the standard construction environment. When the builder needs to run an external tool, such as a compiler, it consults a tool server

that allows for tool execution on various platforms. All tools are executed in an encapsulated environment where all file references are detected and recorded automatically as dependencies for the tool. If any tool is to construct an object already present in the cache, the derived object from the cache is used instead. Vesta's builder performs as well as Make's on scratch builds and significantly faster than Make on incremental builds due to its caching ability.

When provided for use to an engineering group at Compaq, the construction of "wrapper scripts", a domain-specific control panel to construct high level models, and additional system models were necessary to incorporate Vesta into their development environment. Users indicated that these system models became rather complicated and questioned Vesta's usability. It was also stated that for Vesta to be adopted by an organization, there would be a "need to overcome the psychological barrier created by Vesta's radically different approach to SCM".

3.2.3 Shape

Mahler and Lampen [20] developed an SCM system named *Shape* with a significantly enhanced Make program that has access to the version control system and uses configuration rules to identify component versions for build purposes. An attributed file system (AFS) was developed as well to support this functionality. The AFS maintains document attributes that are both inherited from the underlying storage system (like file name, size and owner) and AFS specific (like revision number and

state). Documents are retrieved from the AFS by specifying an attribute pattern.

Shape uses a system description document that consists of four main components: transformation rules, selection rules, variant definitions, and system description. Transformation rules consist of a transformation specification that describes the input and output for the transformation and a transformation script that is passed to a shell process when a transformation is executed. The syntax for these transformation rules is an extension of Make's rule specifications. The selection rules are named sequences of comma separated predicates used to bind concrete document instances to the component names specified in the system description document. The system description component is the same as that for traditional makefiles.

Mahler and Lampen emphasize various incarnations of variants (different versions of the same deliverable) and the difficulty with which they are handled by traditional SCM systems. While the version control system allows for variant identification through use of attributes, their use is not required. These variant attributes can be used in selection rules and passed to transformation tools. Variant classes can be used to define mutually exclusive variant names. Shape can produce a composition list for a given variant, which includes all components contributing to that variant, including tools and environment information involved in its production.

3.2.4 Stellation

Chu-Carrol *et al.* [24] described a general aggregation mechanism that makes use of fine-grained SCM to (1) support multiple overlapping organizations of program source to create virtual source files, (2) allow developers to precisely mark the set of artifacts affected by a change, and (3) associate products from different phases of the development process. They describe aggregation in terms of SCM as "a facility to allow the creation of versioned objects formed from collections of other objects".

Chu-Carrol *et al.* stated that in the aggregate system, relationships between artifacts must themselves be first class artifacts. This requires definitions for various types of relationships in which applicable endpoint types may be specified. The interactions between these aggregate types and the versioning system must be specified as well, particularly in cases where overlapping changes occur. Chu-Carrol *et al.* claimed that the SCM system must provide some mechanism for users to easily search the repository for fragments and use those search results to create versionable aggregate types. Additionally, to take full advantage of the aggregate system, Chu-Carrol *et al.* stated that the SCM system must have some knowledge of the semantics of the artifact types.

An SCM system named *Stellation* was developed that uses an aggregate system like that described and offers method-level storage granularity. Their aggregate types consist of collections of named fields, each of which has a type. Semantic types

are language-dependent and are used to model constructs present in a given language, such as package declarations and class members in Java. Aggregate creation and discovery is possible though queries written in the Stellation Query Language. These queries evaluate annotations written by users either within the artifacts they describe or in separate documents describing those artifacts.

3.2.5 Survey of SCM

Estublier *et al.* [27] wrote about the impact of software engineering research on the practice of SCM. They wrote, "This change from small, simple tools to entire SCM environments can be largely attributed to a steady flow of research, undertaken in both academic and industrial settings, that identified and incrementally improved many ideas, approaches, tools, features, and so on." Estublier *et al.* stated that virtually all major projects use SCM systems, that SCM is essential to the success of any software development project, and that SCM software is now a billion dollar commercial industry. Some of their findings are discussed in this section.

Two of the high-level pieces of functionality provided by SCM systems address the needs of *controlling* and *components*. The controlling functionality provided by SCM is partially described as supporting users in understanding the impact of a change and allowing them to specify to which products a change should apply. The components functionality of SCM is described as supporting users in identifying, storing, and accessing the parts that make up a software system. This functionality

is later related to *data models* and *system models*. Data models and system models aggregate multiple artifacts and the relationships among them into higher-level artifacts which can themselves be versioned.

Research has attempted to define specific data models dedicated to SCM. Many of these data models represent the artifacts and entity relationships of the software system using object-oriented approaches. System models consist of a collection of modules and processes and the relationships between them. As an implementation evolves, the system model must be kept in sync, and vice versa. Additionally, old versions of the models must be maintained so that old versions of the software may be rebuilt.

While much research has been performed on developing more powerful data and system models, it has not had much impact on industrial practice and tools. Estublier *et al.* explained this lack of adoption by saying, "A substantial amount of additional effort is required to define and maintain the system model description. Unless the system model can be automatically updated, the additional effort easily outweighs the expected benefits, especially since compilers catch most interface mismatches. This is why... the use of architecture description languages has not caught on much in industry." Later, they identified three factors that allow the successful impact of an SCM feature. These factors are customer need, ability for developers to provide the needed feature, and ease of use. Additionally, they state, "despite significant potential benefits, most customers will not use a system model if the dependencies among artifacts must be manually specified and maintained". Some

approaches have developed their own build system in which the system model was the central entity responsible for the build process. Many of these systems, however, have since abandoned their approaches in favor of Make.

3.3 Build maintenance tools

This section presents recent research on tools to help users maintain build systems. They include tools that gather their data by observing a Make build, by analyzing a project's makefiles, and by using a custom Make implementation.

3.3.1 MAKAO

Adams *et al.* [8] developed MAKAO, a reverse-engineering framework for build systems. They defined the following functional requirements for MAKAO:

- **Visualisation** - Provide a visual representation of the entire build system
- **Querying** - Support querying for specific information about targets and files involved in the build
- **Filtering** - Allow build data to be filtered from the visual representation
- **Refactoring** - Provide build refactoring operations and update simulations
- **Validation** - Detect and identify bugs in the build system

MAKAO displays a Make build script's dependency graph using color coding, configurable layouts, and zooming. It allows dependency information in the graph to be queried and filtered by writing and executing Gython scripts. It supports refactoring using aspect-oriented techniques, allowing advice to be woven into an existing build script. This advice is implemented using Gython statements that include the text to add to the build script and a description of the locations in the build script to weave it. This refactoring is implemented in memory and can be propagated to the actual build scripts using a Perl script. MAKAO also uses Prolog to validate changes made to refactored build scripts.

MAKAO constructs the build dependency graph using either a modified Make program or by parsing trace output produced by Make. It uses a hybrid of the static and dynamic build analysis approaches. MAKAO begins by analyzing an executed build for a particular configuration and augments this information with static data such as build rules and unevaluated targets.

3.3.2 SYMake

Tamrawi *et al.* [9] developed SYMake, an infrastructure and tool for the analysis of build code in Make. SYMake includes a symbolic evaluation algorithm that produces a symbolic dependency graph (SDG) from a makefile. The SDG represents the build dependencies among files via commands. It differs from a concrete dependency graph of Make in that file names and commands in an SDG may not be completely

resolved into strings. Instead, the SDG's node for a file refers to a V-model, which is a graph-based representation for symbolic string values used to identify a file's name. These symbolic string values may refer to input values or environment data. For each resulting string value in an SDG that represents a part of a file name or a command in a rule, SYMake provides a T-model to represent its symbolic evaluation trace. This T-model shows how those string values are initialized and manipulated via the build process. The SDG generated by SYMake uses static build analysis to provide dependency graphs for various configurations.

SYMake has been used in a tool that can detect several types of code smells and errors in makefiles, as well as to support build code refactoring. Code smells include cyclic dependencies, duplicate prerequisites, and rule inclusion. A cyclic dependency occurs when a target is listed as a prerequisite for one of its prerequisites. Duplicate prerequisites are present when a single prerequisite is listed more than once in a target's prerequisite list. Rule inclusion occurs when a makefile contains a rule for a specific target that is also included elsewhere in a more general rule. Examples of refactoring capabilities include rule extraction and removal, target and variable renaming, and prerequisite extraction.

An empirical evaluation showed that SYMake can achieve high accuracy in entity renaming. This evaluation was conducted using makefiles for seven different software projects. Six Ph.D. students identified the locations in the makefiles that required updating when a given set of variables were to be renamed. SYMake was able to correctly rename 100% of the chosen variables. In contrast, a simple text search

reported a large number of incorrect locations requiring updates.

A controlled experiment showed that SYMake allowed better understanding of makefiles, better code smell detection, and quicker, more accurate refactoring. In this experiment, the makefiles from the empirical evaluation were updated to include code smells detectable by SYMake. Two sets of tasks that involved detecting code smells and refactoring the build were produced. Eight Ph.D. students were divided into two groups. One group completed one set of tasks with SYMake and the other set of tasks without it. The other group completed the opposite set of tasks with and without SYMake. SYMake was shown to achieve significant improvements in both accuracy and effort required (measured in time) to complete the requested tasks.

3.3.3 Amake

Buffenbarger [28] developed a new variant of Make named Amake. File dependencies in Amake are detected, recorded, and monitored automatically. While Make requires a target file's dependencies to be stated explicitly, Amake instead monitors and records the files accessed and programs executed when building a target. It stores this information for all targets in all workspaces on all hosts in a development environment. Additionally, Amake does not rely on operating system timestamps, but rather computes, records, and compares file checksums.

Amake's explicit dependency identification and storage avoids situations in which

a file is referenced in a target's rules but is mistakenly omitted from the target's dependency list. It also tracks dependencies that are not maintained in a traditional Make implementation, such as a target's dependencies on shell-commands, executed programs, shared libraries, and environment variables. These additional dependencies can provide a more accurate indication as to whether or not a target needs to be rebuilt.

3.4 Ant build tools

This section addresses commercial tools to assist in the development of an Ant build system. These tools can be used to simulate the effects of an Ant build, to clarify the high-level behavior of an Ant build, and to manage a project's external dependencies.

3.4.1 Virtual Ant

Virtual Ant [29] is an Ant file creation and maintenance tool that uses a virtual file system to simulate the execution of Ant tasks. It consists of a Windows Explorer-like environment that allows users to create and modify Ant files without directly writing XML. Ant targets and tasks can be created, modified, and rearranged in the build file. The results of executing each task can be seen in the virtual file system displayed by Visual Ant.

3.4.2 Vizant

Vizant [30] is an Ant task that allows users to create an image displaying a build file's target dependency graph. Nodes in the graph represent build targets and edges in the graph represent dependencies between those build targets. If a dependency exists between two targets, then one of those targets must be executed before the other target can be executed. This graph does not, however, reflect the files accessed or generated throughout the build process.

3.4.3 Apache Ivy

Apachy Ivy [31] is a dependency manager that integrates with Ant. With Ivy, dependencies are declared in an ivy.xml file. It is most commonly used to identify external library dependencies, fetch them from a Maven [9] repository, and copy them to a project's lib folder. Ivy can be configured to use other repositories as well. It has corresponding Ant tasks that can be used to retrieve a component's dependencies from a repository as specified in an ivy.xml file. It also produces dependency reports which can be used to generate a graph of a deliverable's external library dependencies, including transitive external library dependencies.

Chapter 4

Implementation

4.1 Introduction

Formiga is a build maintenance and dependency discovery tool for software projects using the Ant build system. It is implemented as an Eclipse plugin, which allows it to recognize when changes have been made to a project and determine if build maintenance is necessary. The remainder of this chapter addresses Formiga's primary features, which are to assist developers with:

- build maintenance due to external changes,
- identification of build dependencies in a software project, and
- build maintenance due to internal changes

4.2 Build maintenance due to external changes

When files in a software project are added, renamed, moved, or deleted, corresponding changes to the project's build files are often required [2, 3]. However, knowing when and where these corresponding changes are needed is not obvious, particularly for developers who don't interact with the project's build system often, or when changes are necessary to a portion of the build system not typically addressed by a developer. This becomes an even bigger problem for projects with large build files or multiple build files.

Formiga is able to recognize when files that have been added, renamed, moved, or deleted affect the behavior of a project's build system. Because Formiga is implemented as an IDE plugin, developers can use the standard Eclipse refactoring operations to add, rename, move, or delete files, which Formiga will recognize automatically using an Eclipse workspace listener. These operations can also be performed directly on the filesystem, and they will be recognized by Formiga as soon as the Eclipse workspace is refreshed. Formiga determines if updates to the build files are necessary to account for the file modifications. It can either make these updates automatically or do so after each update has been confirmed by the user. The confirmation displays the affected target, task, attribute, and old and new attribute values in a confirmation box like those seen when performing typical refactoring tasks.

Formiga aims to make updates to build files that are as "intelligent" as those

```
<property name="docs" value="documentation">
<target name="javadoc">
    <javadoc sourcepath="${src}" destdir="javadoc"/>
</target>
```

Figure 4.1: Property Reuse Example

that would have been made by a good developer. One means of accomplishing this is to reuse property references when updating attributes in the build file. Given an Ant property and target seen in Figure 4.1, suppose a developer uses the IDE to move the "javadoc" directory under the "documentation" directory. Formiga will replace the "destdir" attribute value in the "javadoc" task with "${docs}/javadoc". By maximizing reuse of property values, Formiga retains the high-level logic of the build system.

Formiga's behavior depends on the type of refactoring operation (move, rename, delete, or add), the behavior of the task referring to the refactored file(s), and whether the related references to the refactored file(s) are direct or indirect. A direct reference is one that resolves solely to a single file or directory. An indirect reference is one that may resolve to multiple files or directories. Indirect references include one or more of the following wildcards:

- * - matches zero or more characters
- ? - matches a single character

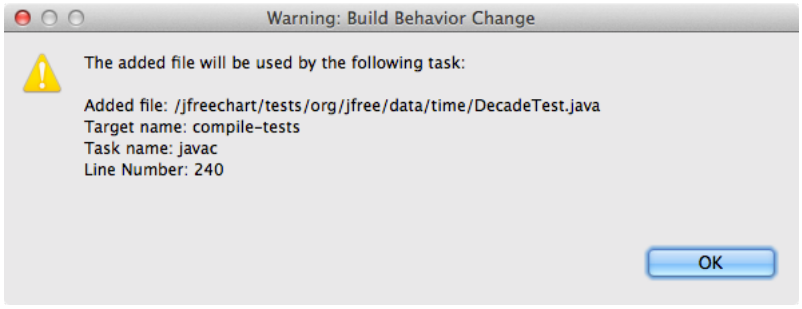

Figure 4.2: Added File Alert

- ** - matches multiple directory levels in a path

4.2.1 Adding a file

If a file is added, Formiga responds by reporting the targets and tasks that will be directly affected by the new file. An example of this alert can be seen in Figure 4.2. A task will be directly affected by the added file if it either operates on all files in the directory to which the file was added, or if it includes an indirect reference that matches the added file. For these cases, updates to the build system are unnecessary, but users are alerted to the effects that the newly added file has on the build system. Although unlikely, if a direct reference to the newly added file was already present in a build file, an alert would be included for the corresponding task as well.

4.2.2 Moving or renaming a file

If a file is moved or renamed, and that file is directly referenced by a task, then that reference will be updated to reflect the new path. This reference must be updated

since it is invalid after the file is moved/renamed. If that file is referenced indirectly by a task and that reference still refers to that file, then no changes will be made to the reference nor will the developer be alerted of any change in the build's behavior. In this case, because the existing indirect reference resolves to both the file's old name/location and its new name/location, the corresponding task operates the same way before and after the file modification. If an indirect reference no longer refers to that file, then either a new reference will be appended to the existing reference (if the original reference was not specified as a nested task), or a new reference will be included as a nested task (if the original reference was specified as a nested task). This will ensure that the modified file is still referenced by the task. Additionally, a moved or renamed file may match an indirect reference that it did not previously match. Like it does for added files, Formiga will alert such cases to the user, so that the user is aware of the tasks that will now use that file as input.

Moving a file from one directory to another may imply that it should no longer be treated the same way by the build system as the files in its previous directory. If this is the case, the user can reject the update and only the existing indirect reference will remain. An example of Formiga's update request due to a renamed file can be seen in Figure 4.3.

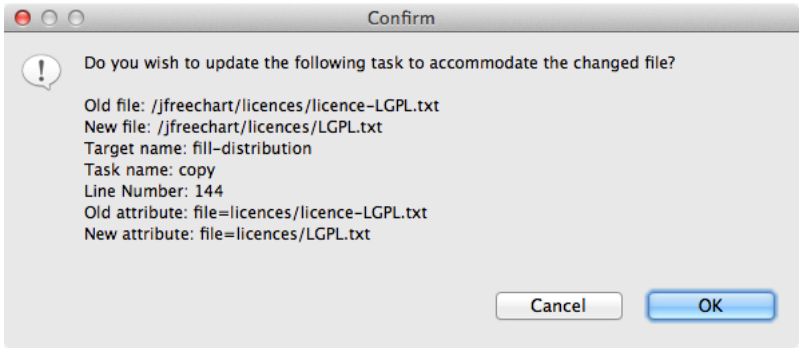

Figure 4.3: Renamed File Confirmation

4.2.3 Deleting a file

If a file is deleted, and the file is directly referenced by a task, then that reference will be removed altogether. If that file was the only file referenced by that task, then that task will be removed altogether, since it no longer has any effect on the build. If that file is referenced indirectly by a task, then no changes will be made to the build system, as the indirect reference may still refer to existing files or directories that may later be populated by files relevant to the task. Again, like it does for added files, Formiga will alert such cases to the user. An example of Formiga's update request due to a deleted file can be seen in Figure 4.4.

4.3 Identification of build dependencies

An Ant build file does not contain (and cannot produce) an explicit identification of the file dependencies for the deliverables it produces. In order to identify the files that a deliverable depends on, one must have a firm understanding of how Ant

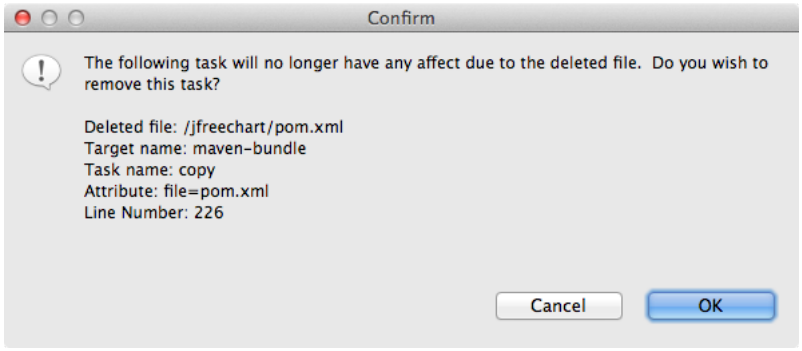

Figure 4.4: Deleted File Confirmation

operates, including the syntax and behavior of all Ant targets, tasks, and properties. For large software projects, these build files can become very complex, thus complicating the task of identifying these dependencies.

We use the term "build dependency" to refer to a dependency between two files that is created by executing a task in a build file. For example, the `javac` task reads java source files and generates corresponding binary class files. The `javac` task creates a build dependency between each source file and its corresponding class file. Each class file depends on the source file that is used to produce it. Similarly, the `zip` task creates a zip archive of a specified collection of files. The `zip` task creates dependecies between the zip file it produces and the files it contains. For our purposes, the zip file depends on the files that it contains.

Formiga identifies these build dependencies for a software project in an IDE and records them in a local Apache Derby database [32], which is automatically installed with the plugin. The process by which these build dependencies are identified is discussed in section 4.3.1. Formiga presents these dependencies using interactive,

directed graphs, which is discussed in section 4.3.2.

4.3.1 Finding build dependencies

This section discusses Formiga's discovery and recording of build dependencies. It describes how and when these dependencies are discovered and recorded as well as how Formiga addresses build configurations.

How build dependencies are discovered

Formiga begins its build dependency discovery by finding the set of targets in a project's build system that no other targets depend on. If each of the targets in this set were executed (which would first execute all of their target dependencies), all targets in the build system would be executed. Essentially, this is producing a set of target chains that span the entire build system. Processing the build dependencies produced during the execution of these target chains ensures that Formiga addresses all possible target chains within the build.

Formiga discovers build dependencies using a modified version of Ant. For each of the targets in the previously mentioned set (or, for all possible target execution paths within the build), Formiga's Ant implementation allows Ant to behave as if it were executing the target, but instead of executing tasks that read and write files to the filesystem, it keeps track of the files accessed by those tasks using a virtual filesystem in memory which we refer to as the "filespace". The filespace maps filesystem locations (both actual and those created during the build) to files

contained at those locations during the build.

The decision to use a virtual fileystem was made for two reasons:

1. Executing tasks that modify the actual filesystem can be time consuming
2. If a task is written incorrectly, it may mistakenly move or delete files

In practice, this means that the files that are read, created, renamed, moved, or deleted by a task are represented in the filespace with an instance of a file model. The file models keep track of various information about a given file, including its location, any files used to generate it (i.e. a java source file is used to generate a binary class file), and any files it may include (i.e. files archived within a zip file). Currently, Formiga models files according to the following file types:

- Source files
- Class files, which can depend on a source file
- External libraries
- Build files
- Deliverables, which can contain instances of all file types
- "Other" files

Additionally, any file model instance can be more generally associated as a dependency of another file model instance. Each file model instance also refers to the

build files that reference them. Support is in place for additional, user-defined file types.

For each task that modifies the filesystem, Formiga must do the following:

1. Identify any files in the filespace used by the task as input
2. Identify any files on the filesystem used by the task as input that have not been "deleted" by a previously "executed" task
3. Add, rename, or delete any files in the filespace that the task would add, rename, or delete
4. Record any dependencies between the task's input and output files

Any tasks that do not modify the filesystem are allowed to execute as they would normally. This is particularly helpful for managing properties used throughout the build.

Each target chain uses its own filespace. However, some target chains may have overlapping subsets of targets. Formiga identifies these and reuses build dependencies identified during the processing of a previous target chain whenever possible. After all target chains have been processed by Formiga, the filespaces are combined to remove redundancies, and the dependencies are recorded to the database. This is discussed in section 4.3.1.

As a result of this implementation, Formiga uses a hybrid of the static and dynamic approaches to build analysis. It is primarily dynamic in nature because

it "executes" an Ant build (albeit without running many of the tools called upon during the build process). However, it uses static data to address the behaviors of the tasks that would otherwise read from and write to the filesystem and to determine which target chains to analyze.

Formiga's build dependency identification was tested using the open source project "Batik" [33], a Java-based SVG toolkit and component of the Apache XML Graphics Project. Its build file contains 2,233 lines of code. Batik was chosen because it is a real software project with a substantial build file, without being so large as to be difficult to work with. Formiga was able to correctly identify and store the build dependencies for each of the six Batik deliverables tested. These deliverables depend on as many as 2,835 other files. These six devlierables were chosen for testing due to the range of tasks involved in their production. The identification of their dependencies includes parsing the following filesystem-modifying Ant tasks: `copy`, `delete`, `jar`, `javac`, `javadoc`, `mkdir`, `move`, `tar`, and `zip`. This set of tasks is enough to support a wide range of build capabilities. Verification was performed by a tool that compared a correct build of each deliverable with builds performed after removing each project resource.

When build dependencies are identified

Build dependencies may be identified and recorded whenever any of the following events occur:

- A build file is modified and saved

- A file is added, renamed, moved, or deleted in the project

Any change to a build file that affects how or when a task is executed during the build may have an effect on the build dependencies it creates, thus requiring a reprocessing of its build dependencies. This includes build modifications made by Formiga when a project has changed. Currently, if a manual build change is made (that is, one that was not made by Formiga), Formiga makes no attempt to determine the nature of the change and reprocesses all of its build dependencies.

If any files are added to the project, renamed, moved, or deleted from the project, and the operation causes the build behavior to change, then the build dependencies need to be reprocessed. Because these operations may first require build maintenance, the determination as to whether or not the build dependencies need to be reprocessed is made when Formiga is checking for build maintenance updates. Currently, Formiga will reprocess all of the build dependencies in the project, however, mechanisms are in place to identify only those targets whose tasks have been updated. Using these identified targets, unnecessary build dependency reprocessing could be minimized.

Recording build dependencies

When all build dependencies have been identified, Formiga records the build dependencies in its local Derby database. Derby [32] is an open source relational database with a small footprint that is implemented in Java. This database is automatically installed with the Formiga plugin. It has access to all projects recognized by the

IDE. To facilitate database interactions, Formiga uses the object-relational mapping tool Hibernate [34]. An object-relational mapping tool allows a class to be mapped to a database table. A class instance can then be saved using functionality provided by Hibernate, causing a record in the corresponding database table to be either inserted or updated. For Formiga, these classes are primarily those used to model files and their dependencies in the filespace.

Before all build dependencies are written to the database, all existing build dependencies for the given project in the database are deleted. The alternative to this requires that every build dependency be checked for existence in the database to determine whether it should be updated or inserted. This process takes longer than simply replacing all build dependencies for the project and has the same effect. The database commit is performed in a separate thread, allowing the developer to continue working with the project while the data is written.

Build configuration handling

Formiga supports dependency extraction for multiple configurations that are implemented using *conditionally set properties (CSPs)*. CSPs are properties that are instantiated only if a specified condition is met. These conditions may include calls to determine the operating system in which Ant is running or evaluating input values when the Ant process is executed.

An example of a CSP named "isMac" can be seen in Figure 4.5. The "isMac" property will be defined if and only if the build is executed in a Mac OS environment

```
<condition property="isMac">
    <os family="mac"/>
</condition>
<target name="buildForMac" if="isMac">
    <zip destfile="prog.zip" basedir="${build}/mac"/>
</target>
```

Figure 4.5: CSP Example

due to the `os` task nested within the `condition` task. The target "buildForMac" can only be executed if the "isMac" property has been defined. Because of its handling of CSPs, Formiga will analyze the "buildForMac" target twice, once with the "isMac" property defined and once without it defined. Formiga will recognize the "isMac" property as a dependency for the deliverable "prog.zip" produced by the target "buildForMac". Besides the `condition` task, CSPs are also created by the `available` and `uptodate` tasks.

When a CSP is encountered by Formiga, it is recognized as such. When that CSP is later referenced within a target, that target is processed twice: once with the property instantiated and once without the property instantiated. If a target references `n` CSPs, then that target will be processed 2^n times. The same is potentially true for any targets that follow in the target chain. Because the same target with different CSP values can potentially produce different files, different filespaces are needed for each processing of that target.

Much like Formiga does when processing multiple target chains, it can reuse

previous filespaces when it recognizes that a previously evaluated target chain with a different set of CSP values produces an equivalent filespace. While this can significantly improve processing time across all occurring CSP permutations, for build systems with a large number of CSPs, this process can be still be time consuming. Additional opportunities for improvement are likely present for this approach. Configuration handling is discussed further in Section 6.3.

4.3.2 Presenting build dependencies

Build dependencies are displayed in Formiga using a directed graph where the nodes represent files and the edges represent dependencies between those files. It allows users to find both *forward* and *backward* dependencies. If file A can only be produced if file B is present (or if file A includes file B), then A is a *forward dependency* of B and B is a *backward dependency* of A. Graphs can be produced for a given project deliverable or for a given project file. Currently, a deliverable graph will display the deliverable's backward dependencies, and a project file graph can display either the file's forward or backward dependencies. Because most project files are not generated from other project files, a project's file forward dependency graph is more useful. These graphs are constructed using the JUNG framework [35]. JUNG is also responsible for the layout of the graph. Formiga's graphs provide the following functionality:

- Highlight a dependency construction location in a build file (via edge click)

- Display backward dependencies for a file (via node click)
- Filter by file name, path, and/or file type
- Zoom in/out
- Show/Hide file names
- View file path (via node hover)
- Manual node rearrangement

When Formiga discovers dependencies in a build, it records the line numbers in the build file that are responsible for the construction of those dependencies. When an edge is clicked in the graph, this information is used to open the build file in the IDE and highlight the location where the indicated dependency is constructed. Without Formiga, this is not a simple task, particularly for projects with a large build system. A deliverable's dependency graph will automatically display backward dependencies for all included files, but the ability to display a file's backward dependencies when its node is clicked is useful after filtering. This functionality is also useful for graphs displaying a file's forward dependencies. Filtering and zooming are particularly useful for deliverable dependency graphs, since they are more likely to contain a large number of nodes than dependency graphs for project files. File names are not shown automatically, as they may be distracting in graphs with many nodes, so the ability to show and hide file names is provided. Lastly, a file's

path can be seen when the cursor hovers over the file's corresponding node in the graph, and graph nodes can be rearranged.

Formiga is designed to integrate into an IDE, leveraging existing metaphors used to maintain source code. Developers are accustomed to accessing various source code dependency and refactoring operations using context menus associated with project resources and source code units. Formiga mimics this behavior by adding context menu items to project resources and Ant build file components. A "Formiga" option is added to the package explorer context menus for a project and its files, which allows a graph to be generated for a specific project deliverable or file.

Dependency graphs for a given deliverable produced by a project's build system can be generated by selecting the project's "Formiga" context menu item and choosing the desired deliverable, as seen in Figure 4.6. An example of a backward dependency graph for a project deliverable can be seen in Figure 4.7.

Dependency graphs for a given file in a project can be generated by selecting the file's "Formiga" context menu item and choosing either "Forward Dependencies" or "Backward Dependencies", as seen in Figure 4.8. An example of a forward dependency graph for a project file can be seen in Figure 4.9.

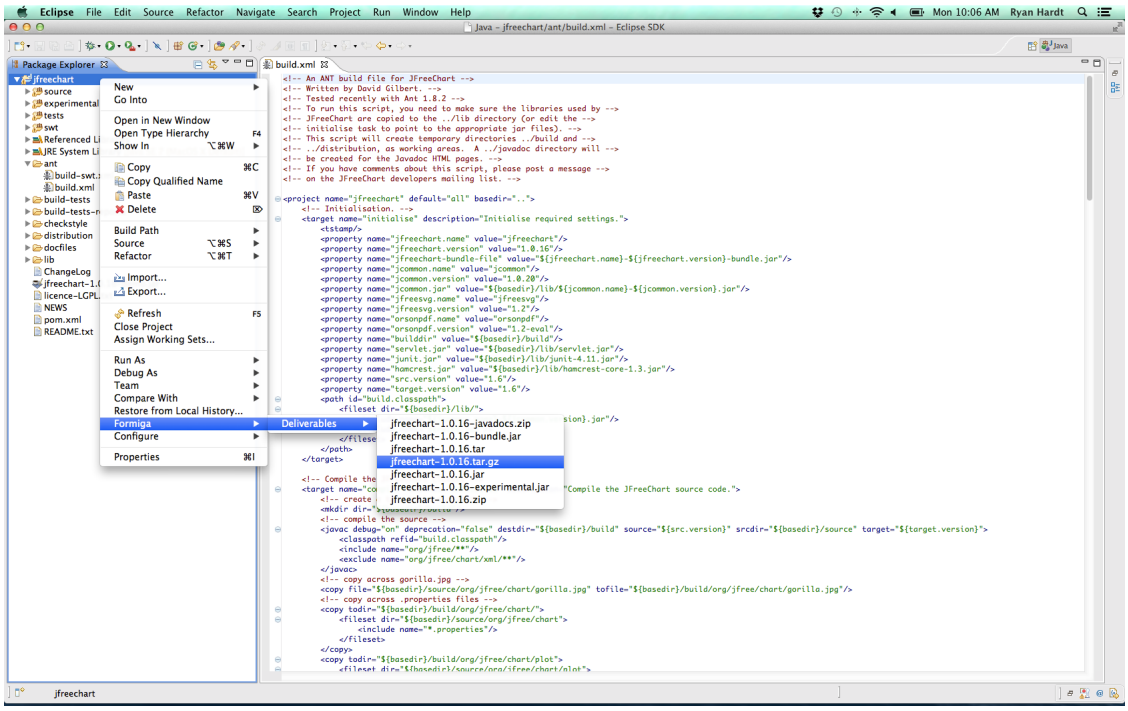

Figure 4.6: Project Deliverable Dependencies

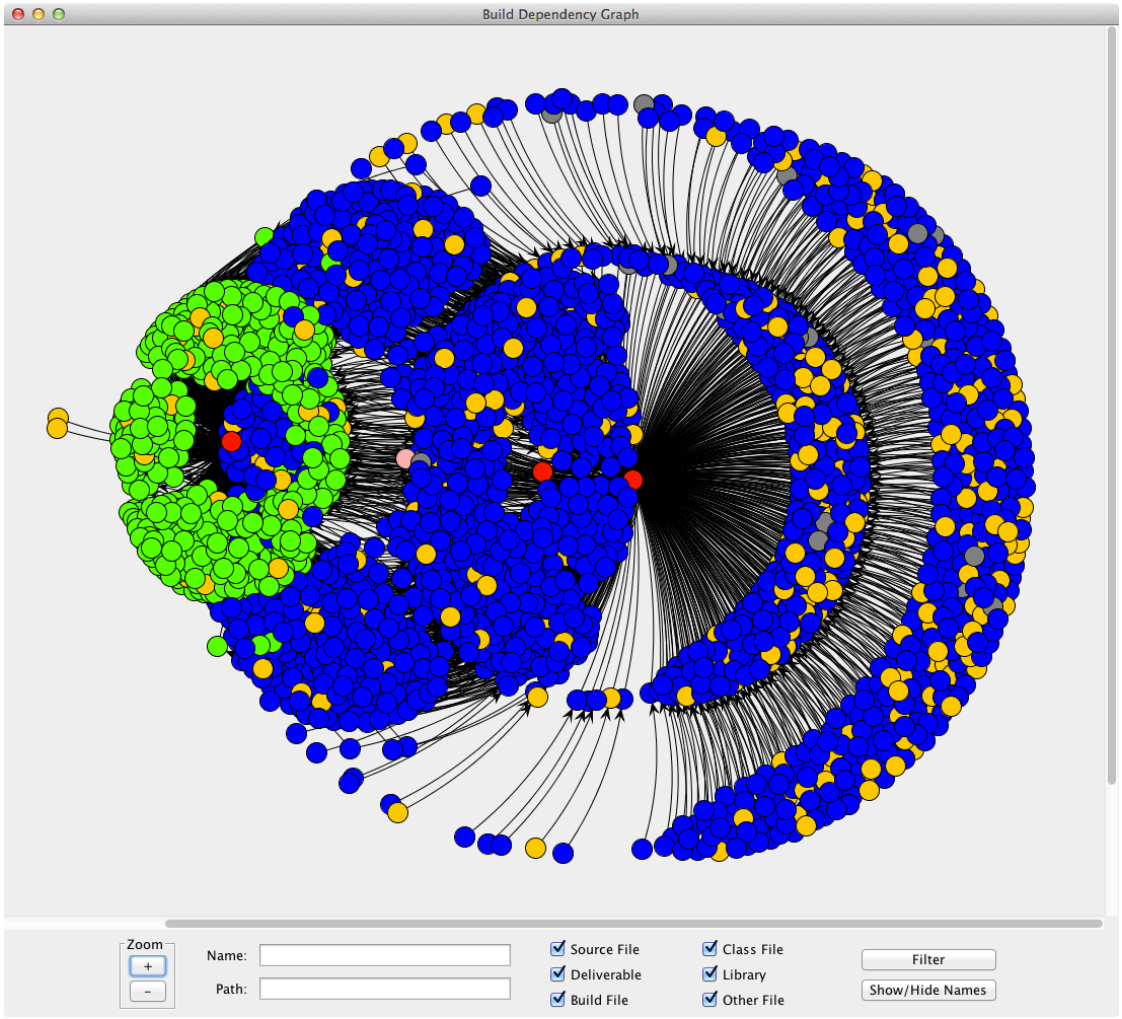

Figure 4.7: Deliverable Dependency Graph

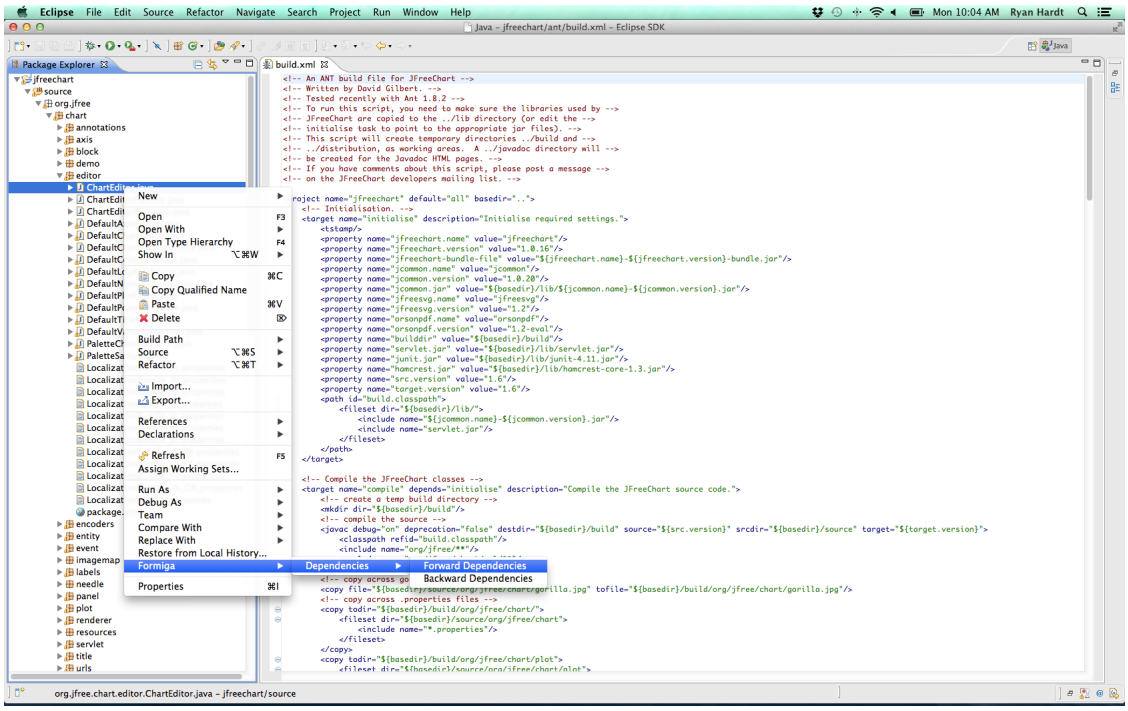

Figure 4.8: Project File Dependencies

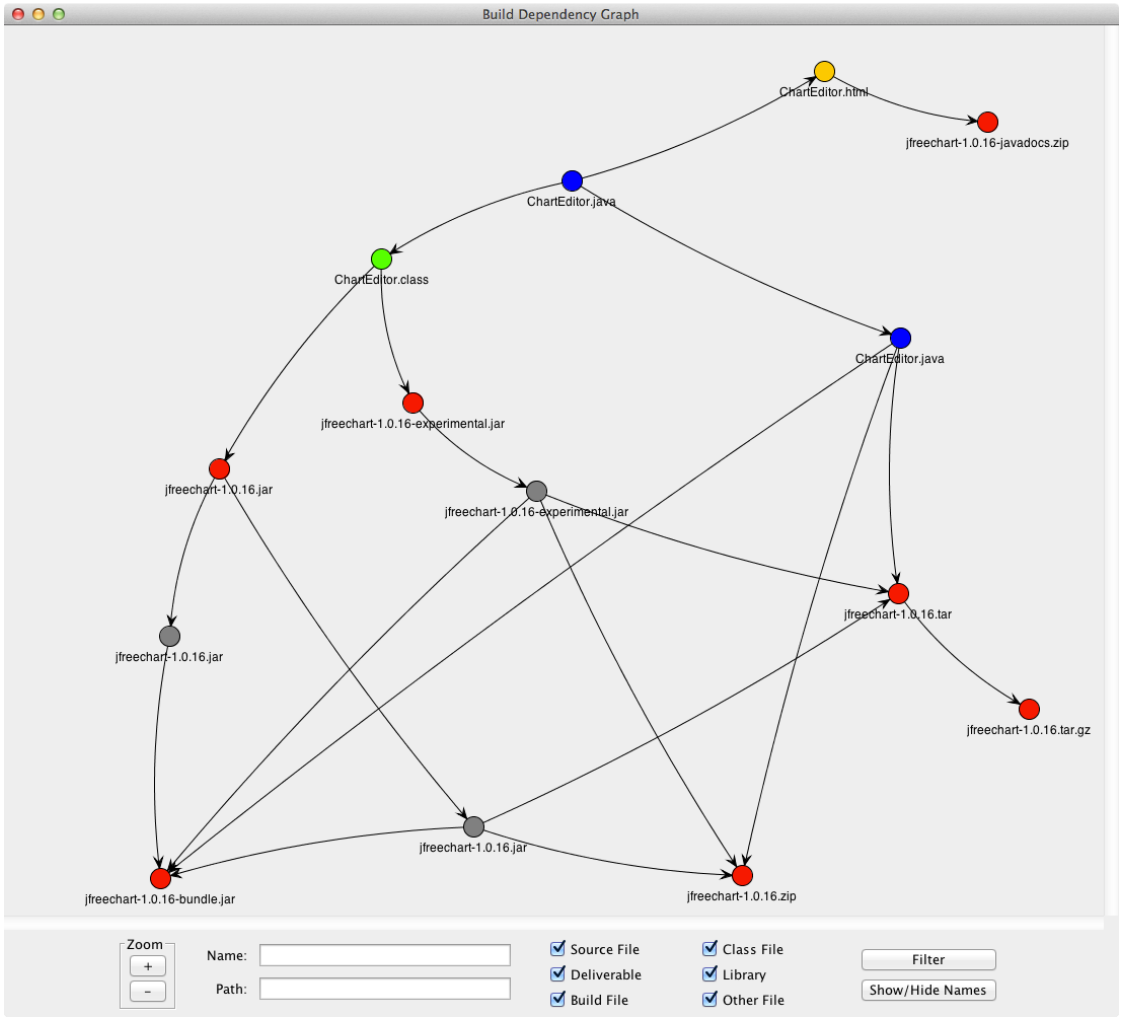

Figure 4.9: Project File Forward Dependency Graph

4.4 Build maintenance due to internal changes

Formiga allows developers to make changes directly to the build system more easily by renaming and removing targets and properties. These build refactoring operations can be error prone if performed manually, particularly if they require a large number of updates or span multiple build files. Support for these operations within IDEs is limited to a basic find and replace for all document text, which can indicate many false positives.

Much like it does for displaying build dependencies, Formiga leverages metaphors used to refactor source code to facilitate build refactoring. To refactor variables and methods in source code, users can highlight the name of a variable or method and select a desired refactoring operation within an associated context menu. Formiga adds context menu options to highlighted build targets and properties to allow build refactoring. An example of the build refactoring context menu can be seen in Figure 4.10.

4.4.1 Target removal

To remove a target with Formiga, users can highlight the name of the target at its declaration, and select "Remove Target" from its context menu. Formiga will remove the target from the build file as well as any references to that target. These references could appear in the dependency lists of other targets or as an attribute value in the `antcall` task. If the removed target depends on another target that

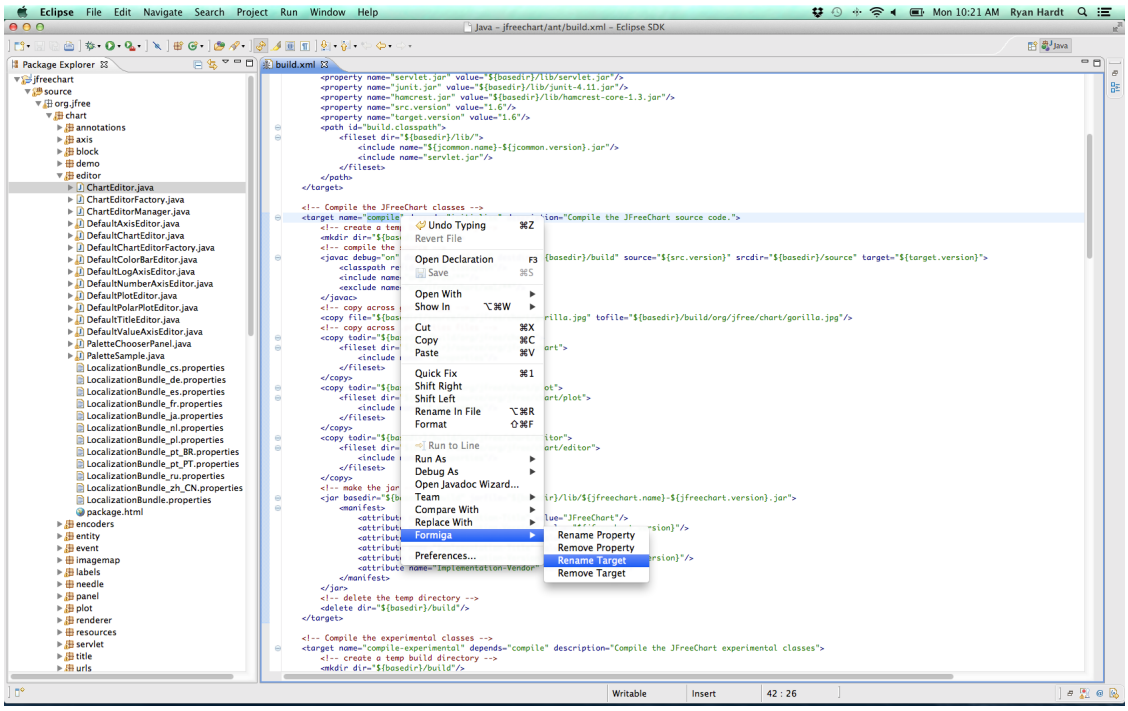

Figure 4.10: Build Refactoring Context Menu

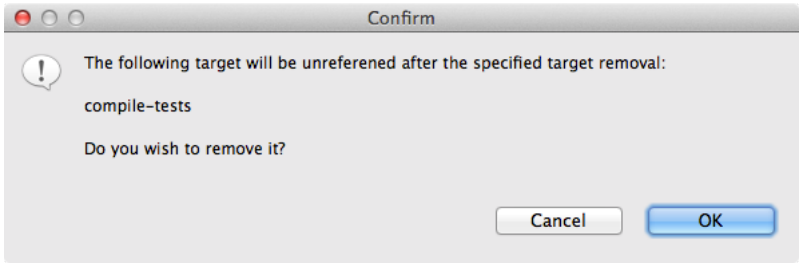

Figure 4.11: Target Removal Confirmation

is unreferenced elsewhere in the build, Formiga will prompt the user for removal of that target as well, as seen in Figure 4.11. This practice avoids the presence of dead code within the build system. After the refactoring as been completed, Formiga will report the number of removed targets and references.

4.4.2 Target renaming

To rename a target with Formiga, users can highlight the name of the target at its declaration, and select "Rename Target" from its context menu. Formiga will prompt for a new target name and replace the existing target name and all references to it with the new name. After the refactoring as been completed, Formiga will report the number of updated target references.

4.4.3 Property removal

To remove a property with Formiga, users can highlight the name of the property at its declaration, and select "Remove Property" from its context menu. Formiga will remove the property from the build file and replace any references to that property

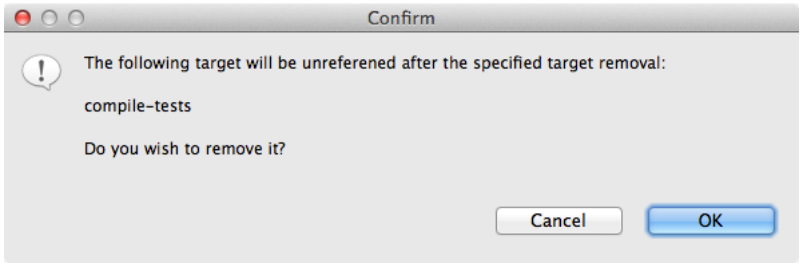

Figure 4.12: Property Rename Alert

with its previous value. This will ensure that the build behaves the same way before and after the property removal. These references could appear nearly anywhere in the build. After the refactoring as been completed, Formiga will report the number of updated property references.

4.4.4 Property renaming

To rename a property with Formiga, users can highlight the name of the property at its declaration, and select "Rename Property" from its context menu. Formiga will prompt for a new property name and replace the existing property name and all references to it with the new name. After the refactoring as been completed, Formiga will report the number of updated property references, as seen in Figure 4.12.

Chapter 5

Controlled Experiment

5.1 Organization

A controlled experiment was conducted to assess Formiga's capabilities. The study was conducted primarily to answer two research questions:

1. Does Formiga decrease the time required to maintain a build system?
2. Does Formiga improve the accuracy with which users can maintain a build system?

The study was conducted using the Java software project "JFreeChart" (version 1.0.16) [36]. This project was chosen for the following reasons:

- It is open source
- It uses the Ant build system

- Its build is well organized
- Its build file's size is appropriate for inexperienced Ant developers (389 lines)
- It produces an appropriate number of deliverables for the study (6 deliverables)
- The total project size is appropriate for the study (1143 files)

JFreeChart contains two build files, neither of which references the other. For the sake of the inexperienced build developers, only the main build file was addressed in this experiment. No changes were made to this project for the purposes of this experiment, other than to the formatting of the main build file. Formiga does not require that an Ant build file is formatted in any particular way, but the build files it produces when it makes modifications have a fixed format in which nested elements are always indented using a single tab.

The experiment was run in a conventional university office that happened to be out of active use during the period of the study. The experimenter was present in the room with the subjects during the study. Subjects performed the experiment's tasks using a Macbook Pro laptop with a 2.66 GHz Intel Core i7 processor, 4 GB of ram, and the OS X Mavericks operating system. A wired mouse was available for use. Because Formiga is integrated with the Eclipse IDE (version 3.8), this IDE was used for the study.

5.1.1 Subjects

Sixteen subjects participated in the experiment. Subjects were chosen from a pool of former coworkers, acquaintances, graduate students, and undergraduate students. Before a subject was chosen to participate, that subject was asked about his or her knowledge of Ant and/or Make. Subjects were required to either have familiarity with Ant or express comfort with Make. Potential subjects were also asked to specify their level of knowledge of Ant, so that we could obtain 8 subjects with little-to-no knowledge of Ant and 8 subjects with at least a moderate knowledge of Ant.

5.1.2 Experimental Procedure

Each subject was required to first read and agree to the consent letter, which can be seen in Appendix A. If needed, subjects were provided instruction on using the Eclipse IDE and Mac OS X. Subjects were also given an overview of Ant if needed. An explanation of the JFreeChart project and its build system were provided to all subjects. The purpose of each target in the project's main build file was discussed. Subjects were given the opportunity to ask questions about JFreeChart and its build system after they were described.

The experimental procedure was designed to test the speed and correctness with which subjects could perform eleven (11) build maintenance and refactoring tasks, both using Formiga and not using Formiga. The study used a within-subjects design, so each subject actually performed 22 *concrete tasks*, two for each of the eleven

abstract tasks. The experimental session was divided into 3 sections. Each section consisted of a series of tasks and/or questions about the software project and its build system. The first section contained 8 concrete tasks and addressed build maintenance due to external changes. The second section contained 6 concrete tasks and addressed how project resources contribute to the deliverables produced by the build system. The third section contained 8 concrete tasks and addressed build maintenance due to internal changes. Before each section, subjects were given a demonstration of Formiga's related capabilities as well as how those capabilities could be performed without using Formiga. Subjects were given the opportunity to ask questions about any other relevant functionality of the IDE.

In order to avoid biased results due to order effects, the study used a balanced design relative to both the order in which subjects performed the two concrete tasks corresponding to a particular abstract task and the order of the experimental (with Formiga) and control (without Formiga) conditions. Furthermore, the design ensured that order balancing was not confounded with the subjects' level of Ant expertise.

For each concrete task, subjects were given a narrative describing a hypothetical situation that required a modification of the build file and/or a question to be answered about the build. These narratives can be seen in Appendix A. Before each subject began a task, he or she was allowed to ask clarification questions about the provided scenario. Subjects were also told that questions asked while the tasks were being performed might not be answered if the answer provided too much insight

into the task. The subjects were instructed to indicate when they were ready to begin the task as well as when they had finished the task. Each task was timed and the modifications and/or answers recorded. Modifications and/or answers were identified as either correct or incorrect. Upon completion of the study, subjects were given a survey, which can also be seen in Appendix A. The remainder of this section describes the tasks performed in each section of the experiment.

Build maintenance due to external changes

This section addressed changes to the build system when project resources were added, renamed, moved, and deleted. Subjects were given a demonstration of each operation using Formiga and were also shown how this information could be obtained without Formiga for the same modifications. For the tasks performed using Formiga, subjects actually added, renamed, moved, and deleted the indicated project resources. For the tasks performed without Formiga, subjects did not actually add, rename, move, or delete the indicated project resources but were instead asked to anticipate what effects it would have on the build system. Because of these slight variations in operations, two versions of each narrative in this section were written, one in which the project resources were actually modified, and the other in which the project resources were not actually modified. In the latter case, if those operations were performed, Formiga would have informed the subjects of their effects on the build system.

Because this was the first section performed, subjects were likely to spend more

time on the first few tasks to better familiarize themselves with the build. To account for this, half of the subjects addressed the tasks in the following order: adding files, renaming files, moving files, and deleting files. The other half addressed the tasks in the opposite order.

For the two tasks regarding added project resources, subjects were asked to identify tasks that were "directly affected" by the added files. For the purposes of this study, a task is "directly affected" by an added file if it would use the file (at its original location) as input. For these tasks, subjects were asked to write down each target and task that was directly affected by the added file.

For the four tasks regarding renamed and moved project resources, subjects were asked to modify the build so that it operated the same way after the rename or move as it did before the operation. In other words, all tasks were asked to have the same set of input files before the operation as they did after the operation. If using Formiga, subjects were asked to write down the modified targets and tasks. If Formiga was not used, subjects were asked to make the necessary changes to the build file.

For the two tasks regarding deleted project resources, subjects were asked to make any necessary changes to the build to account for the deleted file as well as to identify any tasks that would no longer use the file as input, but would not require modifications. The former identifies tasks containing direct references to the deleted file, and the latter identifies tasks containing indirect references to the deleted file. If using Formiga, subjects were asked to write down this information. If Formiga

was not used, subjects were asked to make the necessary changes to the build file and to write down any targets and tasks that would no longer use the file as input but did not require modifications.

Deliverable construction

This section addressed how project resources contribute to the deliverables produced by the build system. Subjects were given a demonstration of Formiga's ability to generate dependency graphs for files in the project as well as for the deliverables produced by the build system. The graph's filtering operations and construction location identification were demonstrated as well. Subjects were also shown how one would trace these same dependencies without Formiga. Subjects were not given any further instruction on how to generate or use the graph after the initial demonstration.

For the first two tasks, subjects were asked to identify all deliverables to which a given project resource contributed. A list of all deliverables produced by the build system was provided in the narrative. Subjects were asked to write down the names of each deliverable whether or not Formiga was used for the task.

For the next two tasks, subjects were asked whether or not the contents of a given directory were involved in the construction of a specified deliverable. Subjects were asked to simply provide a "yes" or "no" answer to these questions.

For the last two tasks in this section, subjects were asked to identify all targets

and tasks involved in a given project resource's contribution to a specified deliverable. Subjects were asked to write down these targets and tasks whether or not Formiga was used for the task.

Build maintenance due to internal changes

This section addressed changes made directly to the build system when renaming and removing properties and targets. Subjects were given a demonstration of Formiga's ability to perform each of these tasks along with a description of the required changes without using Formiga. For all tasks performed with Formiga, subjects simply indicated when they had finished the task. For all tasks performed without Formiga, subjects were asked to make the necessary changes to the build file and indicate when they had finished. Correctness was determined either while the task was being performed, after the task had been completed, or after the subject had completed the experiment. The resulting build file for each subject was saved.

Subjects were first asked to rename two properties from the build file. Next, two build targets were renamed. Subjects were then asked to remove two properties from the build file, replacing their references with their previously specified values. Lastly, subjects were asked to remove two targets from the build file. Subjects were told that if the target removal resulted in unreferenced targets, those unreferenced targets should be removed as well.

5.2 Experiment results

Times to complete each task with and without Formiga were tested for statistical significance using a paired, two-tailed, Student's t-test with 15 degrees of freedom. The subjects' average task completion time (both with and without Formiga) and paired-samples t-test values can be seen in Table 5.2. A paired-samples t-test was used because, for each row in the table, subjects completed two tasks: one with Formiga and the other without it. The t-test values show that, for nearly all tasks, Formiga has a statistically significant impact on the time required to complete the build maintenance tasks performed. For example, the subjects' times to complete the "Add File" task with Formiga were significantly faster than their times without Fomriga ($t(15)=6.83$, $p < .0001$). Tasks for which the p-values are less than .05 indicate that there is less than a 5% chance that randomly generated data would produce the same results. In other words, for our purposes, the p-value indicates the probability that there is no real difference between completing the corresponding task with and without Formiga.

The completion time results indicate that Formiga saves developers time in all 3 categories of tasks: those related to build maintenance caused by external changes, those focused on deliverable construction, and those related to build maintenance caused by internal changes. When performed without Formiga, the tasks related to project changes and deliverable construction are time consuming due to their complexity (this is demonstrated in the correctness results). While less complex than

Task	Time(s)		t(15)	p
	Formiga	Without		
Add File	38.75	195.13	6.83	<.0001
Rename File	34.81	97.50	5.13	<.001
Move File	32.38	184.19	7.10	<.0001
Delete File	52.63	186.31	4.73	<.001
Forward Dependencies	95.56	290.38	6.23	<.0001
Backward Dependencies	102.94	123.13	0.87	n.s.
Dependency Construction	125.19	202.13	2.59	<.05
Rename Property	27.19	92.69	3.69	<.005
Rename Target	23.56	60.56	8.69	<.0001
Remove Property	20.63	132.31	5.03	<.001
Remove Target	34.69	112.75	7.00	<.0001

Table 5.1: Task completion time

the other tasks, the build refactoring tasks are also time consuming when performed without Formiga, as they often require one to examine each token identified by a basic text search to ensure its appropriateness for updating.

The only task for which this p-value was greater than .05 was the "Backward Dependencies" task. For this task, subjects were asked whether or not the contents of a directory contributed to a deliverable. The most efficient way to accomplish this task with Formiga is to produce the backward dependencies for the indicated deliverable and filter the results using the directory's path. However, many subjects instead produced the forward dependencies for each file in this directory, checking to see if the indicated deliverable was present. This approach is more time consuming than the expected approach, significantly increasing the time required to complete the task.

Correctness was tested using a Pearson's chi-squared test with one degree of

freedom. The subjects' task correctness (both with and without Formiga) and chi-squared test values can be seen in Table 5.2. The chi-squared values show that, for nearly all tasks, Formiga has a statistically significant impact on the accuracy with which build maintenance tasks can be performed. For example, the subjects were more able to correctly identify affected tasks in the "Add File" task with Fomriga than without it (χ^2=10.67, p <.001).

The correctness results indicate that Formiga is most useful for automatically updating the build files when project resources are changed. This is especially true for tasks that required subjects to identify how project files were used by the build system. This was a primary component of the "Add File", "Move File", "Delete File", and "Forward Dependencies" tasks. While still significant, Formiga has less of an impact on the correctness of the tasks that required users to rename and remove build targets and properties. These tasks are error prone but require less understanding of the responsibilities of a project's build system than the other tasks.

The only tasks for which this p-value was greater than .05 were the "Backward Dependencies" task and the "Rename Target" task. The "Backward Dependencies" task had a yes or no answer. All of the subjects who answered this question incorrectly using Formiga used the filtering operations incorrectly. The "Rename Target" task was likely the most straight-forward build refactoring task in our experiment, thus diminishing the benefits provided by Formiga for this purpose.

Task	Formiga		Without		χ^2	p
	C	IC	C	IC		
Add File	16	0	8	8	10.67	<.001
Rename File	16	0	11	5	5.93	<.01
Move File	16	0	4	12	19.20	<.0001
Delete File	16	0	6	10	14.55	<.0001
Forward Dependencies	12	4	1	15	15.68	<.0001
Backward Dependencies	12	4	10	6	0.58	n.s.
Dependency Construction	9	7	3	13	4.8	<.05
Rename Property	16	0	13	3	3.31	<.05
Rename Target	16	0	14	2	2.13	n.s.
Remove Property	16	0	13	3	3.31	<.05
Remove Target	15	1	11	5	3.28	<.05

Table 5.2: Task correctness

5.3 Survey results

Upon finishing the experiment tasks, subjects completed a survey, the results of which can be seen in Appendix A. This survey included questions about Formiga's usefulness, subjects' levels of experience, and general likes/dislikes regarding Formiga.

5.3.1 Formiga's usefulness

As a part of this survey, subjects were asked to rate Formiga's usefulness for each task performed, according to the following evaluation scale:

(5) Excellent (4) Very Good (3) Good (2) Fair (1) Poor

All subjects responded very positively to Formiga. Their responses to this portion of the survey can be seen in Table 5.3.

Task	Average
Add File	4.63
Rename File	4.94
Move File	4.94
Delete File	4.81
Forward Dependencies	4.44
Backward Dependencies	4.63
Dependency Construction	4.50
Rename Property	4.81
Rename Target	4.88
Remove Property	4.69
Remove Target	4.75
Overall	4.88

Table 5.3: Survey responses on Formiga's usefulness

5.3.2 Subject levels of experience

The survey also included questions regarding the subject's development experience and knowledge of Ant. Subjects were asked to identify themselves as either professional developers (7 out of 16) or students (9 out of 16). Those identifying themselves as professional developers reported an average of 6.1 years of professional development experience. Those identifying themselves as students (many of whom also indicated some level of professional development experience) reported an average of 9.67 semesters of studying computer science.

All participants were asked about their experience with Ant. Subjects reported an average of 2.53 years of experience of Ant. Subjects were also asked to indicate their level of knowledge of Ant using the following scale:

None Little Moderate Experienced Expert

Three subjects reported "None", five subjects reported "Little", six subjects reported "Moderate", and two subjects reported "Experienced".

5.3.3 Formiga likes/dislikes

Lastly, the survey included questions regarding what subjects liked best about Formiga, liked least about Formiga, and what capabilities were missing. Selected responses have been included below along with a more general representation of user responses.

What did you like best about Formiga?

Five subjects indicated something specifically about the automatic build updates when refactoring project resources. An additional six subjects made more general comments about automatic updates to the build file. Six subjects referred to the dependency graph.

- "Can see a lot of search time being cut from build manipulation. Effortless source reorganization without worrying you broke the build."
- "That refactoring of dependencies in the build was automatic. I realize thats the purpose of the application, but it really makes it almost too easy to modify the build based on refactoring code."
- "Automation of lot of manually intensive tasks like checking dependencies, etc. The colorful representation of deliverables and its dependent files."

- "Reduced trial and error when it comes to maintaining projects. Dependency graph edges showing lines in the build XML was extremely useful."
- "Eliminates possibility of subtle refactoring errors when performing a task by hand"

What did you like least about Formiga?

Four subjects mentioned a lack of feedback. Four subjects reported something about the dependency graph.

- "The graphing was a little confusing at first, but after a bit of time, it would make perfect sense."
- "Maybe some sort of logging would be nice so you can see all that has been changed. Otherwise you only see one popup telling you what it did and that's all"
- "The graphing functionality was very pretty but somewhat hard to use."
- "Did not show where refactorings were occurring when renaming or removing properties and tasks"
- "Minor lack of feedback for refactoring"

Are there any capabilities that you wish Formiga had?

Two subjects mentioned additional feedback (both of whom also reported this for the previous question). Two subjects also expressed a desire for another representation of the build.

- "Build diagram showing a nicely formatted list of properties/targets with descriptions/dependencies."
- "More detail/specifics regarding targets/tasks affected by rename and deletes."
- "Maybe a UI on top of the build xml with all your targets so you don't have to scroll through to find the one to modify/delete"
- "When determining if any file in a directory is used by another deliverable, 'Check all in directory' graphing functionality would be nice."
- "Removing related comments."

Chapter 6

Constraints

6.1 Implementation

Formiga assists in build maintenance for software projects using the Ant build system. It does not currently work with other build systems. Additional build system support is described in Section 8.2. Formiga is implemented as an Eclipse plugin and has been tested with version 3.8 (Juno) of Eclipse. Due to its use of IDE specific code, Formiga is unlikely to work as-is with other versions of Eclipse. Support is planned for integration with more recent versions of Eclipse.

6.2 Task support

Formiga uses knowledge of tool behaviors and Ant task specifications to determine the build dependencies imposed by those tasks and tools. The `exec` and `java`

tasks execute a user-specified program or Java class. We refer to these tasks as *arbitrary execution tasks* or *AET*s. Because Formiga cannot predict the behavior of AETs, they also cannot be processed in the same way as the other packaged Ant tasks. Additionally, Ant allows for the implementation of custom tasks. Like AETs, Formiga cannot predict the behavior of custom Ant tasks, so it cannot include them in its dependency identification. However, if formatted comments describing these tasks were present in the build file, the dependencies created by those tasks could be included in Formigas analysis. These formatted comments could consist of a comma-separated list of input and output files read and written by the task. The same property references and wildcard patterns recognized by Ant could be used to identify files. Formiga could then (at least) describe the tasks input files as dependencies for the task itself and the task as a dependency for the output files it produces.

6.3 Configuration support

Configurations are recognizable by Formiga as long as they are specified using conditionally-set properties (CSPs). For projects with a large number of frequently referenced CSPs, dependency identification could take a considerable amount of time. While this process can be executed in a separate thread allowing the user to continue working, multiple build updates in succession or a save upon closing the IDE could cause undesirable wait times for the user. A significant amount of effort

has been spent on reducing this time, but more evaluation is necessary.

Another common way to identify configurations in an Ant build is through the use of build property files. These property files define a set of Ant properties and values that can be referenced by build files. Typically, a property file exists for each desired configuration or environment, and each property file provides values for the same set of properties. While Formiga does not currently support configurations using build property files, support for their use is planned. Users could identify the location containing these build property files, and Formiga could evaluate the build once for each property file.

6.4 Dependency identificaiton

Formiga identifies files both on the filesystem and in its filespace by their name and location. This means that, if a file with the same name and location is produced by the build system by multiple targets, it is considered to be the same file. If that file's dependencies are different based on which target is producing it, those dependencies will be merged by Formiga. Currently, Formiga assumes that this is bad practice, and while it hasn't been seen in any builds evaluated thus far, more consideration is necessary to determine whether or not Formiga's behavior here is truly appropriate.

Chapter 7

Contributions

Formiga's most novel feature is its ability to automatically update build files when project resources are refactored. This functionality automates many of the build updates necessitated by changes to a software project. Because these operations often require accompanying build maintenance, it saves developers time and ensures that the build isn't broken to due to a neglected or incorrect build update. Furthermore, it informs developers of behavioral changes in the build even when no build content changes are necessary, thus increasing the visibility of the build during routine project refactorings.

Formiga's unique approach to build analysis allows the benefits of dynamic analysis without all of the costs. It runs more quickly than a traditional dynamic approach, because it allows Ant to behave as if it were executing tasks without any unnecessary tool execution for analysis purposes. This approach also won't produce any undesirable side effects caused by destructive build operations, since it does

not modify the filesystem. Formiga's analysis approach is possible because the Ant tasks that Formiga models have fixed semantics, making their behavior predictable.

Our controlled experiment analyzed the ability of developers to perform build maintenance tasks that accompany project changes both with and without Formiga. These tasks included identifying build operations affected by project changes, updating the build to account for project changes, and identifying how project files are used by the build system. We are unaware of any other experiments that assess the abilities of developers to perform these tasks. Our experiment demonstrated that Formiga decreases the amount of time necessary to perform build maintenance while improving correctness. Additionally, although a more exhaustive study would be needed, this experiment suggests that developers have a difficult time identifying exactly how a build system uses the files on which it operates.

Formiga's implementation as an Eclipse plugin allows it to leverage existing metaphors that developers use to maintain source code. This is possible because many build maintenance operations are similar to source code maintenance operations. Formiga is intended to be practical and easy to use. Its implementation as a plugin is a key component in facilitating both characteristics. The results of the controlled experiment are largely a testament to Formiga's usability. Furthermore, for development teams in which build maintenance is dispersed amongst the developers (rather than using a dedicated team for build maintenance), the integration of a build maintenance tool with their existing toolset is likely to be a major benefit, as build maintenance has been shown to be coupled with source code maintenance.

Chapter 8

Future Research

8.1 Repository integration

Initial support is in place to integrate Formiga with a software repository. Doing so would allow Formiga to version the build dependencies that it identifies. In our design, an instance of Formiga residing with the software repository would identify and record backward dependencies of project resources as they are committed to the repository, much like it does currently when files are saved locally in the IDE. Formiga could then version these dependencies using the same version identifiers used by the version control system (VCS) to identify the file versions. Using this data, Formiga would be able to produce dependency graphs for prior versions of a software project. Additionally, Formiga could allow build dependencies to be compared between components recognized by the IDE and components of prior versions

recorded by the VCS. Versioning build dependencies would make it easier for developers to understand differences between the build systems of different versions of a software project. Without this functionality, it may be necessary to manually compare different versions of one or more build files (as well as the resources they operate on) or evaluate multiple versions of a software project using a build maintenance tool.

8.2 Integration with other build tools

In addition to assisting developers with build maintenance for software projects using the Ant build system, it would be interesting to see if Formiga can provide similar benefits for projects using Maven and/or Gradle. Because both build systems are packaged with the tools that they use to build software projects (and because they are both open source projects), an analysis similar to Formiga's analysis of an Ant build system could be performed. To integrate either build system with Formiga, the build system's source code would need to be modified similarly to Ant's. This would require adding hooks into the source code responsible for executing the tools used throughout the build process.

Because Maven takes a "build by convention" approach to building software projects, it essentially shields developers from the tools it uses during the build process. Perhaps more so than any other build system, the required understanding of a software project's Maven build system is likely to differ between the developers

who are responsible for creating a project's Maven build versus the developers who simply execute Maven's build operations for a project. It would be interesting to see what benefits Formiga could provide to both sets of developers.

Because of its emphasis on flexibility, the Gradle build tool is relatively similar to Ant. However, due to its implementation in Groovy, the learning curve for Gradle integration is likely higher than it would be to integrate Maven. Also because of its flexibility, Gradle is packaged with many tools that can be used by a project's build system, requiring substantially more effort to integrate. Because it allows for Ant tasks to be directly executed within a Gradle build file, much of the work to analyze Ant builds could be reused for Gradle. Since, like Ant, many Gradle builds are procedural, Formiga may provide similar benefits for Gradle users as it does for Ant users.

Chapter 9

Conclusion

Despite the demonstrated need for build maintenance tools due to the growth and complexity of build systems, few such tools exist. Formiga addresses this need by offering a build maintenance and dependency discovery tool for software projects using the Ant build system.

Formiga automates build maintenance due to external changes, alerts developers when project modifications will affect the build, and facilitates build maintenance due to internal changes. It also aids in the understanding of a project's build system by identifying how a project's files are used by the build system using an interactive graph.

Our controlled experiment demonstrated that Formiga does indeed allow users to perform build maintenance and identify build dependencies more quickly and accurately than is possible without it.

Formiga is intended to be unobtrusive and intuitive. It is distinguished by its

ability to automatically update or inform developers when project changes require accompanying build changes, its unique analysis approach to discover build dependencies, and its implementation as an easy-to-use Eclipse plugin.

Appendix A

Controlled Experiment Files

Informed Consent
UW - Milwaukee

IRB Protocol Number: 13.427

IRB Approval date: June 14, 2013

Dear participant,

You are invited to participate in a research study, entitled "Evaluating the effectiveness of Formiga on build maintenance". The study is being conducted by Ryan Hardt and Ethan Munson of the University of Wisconsin - Milwaukee.

The purpose of this research study is to evaluate the usefulness of a build maintenance tool when updating the build system in response to source code changes, performing build maintenance (such as modifying/removing build targets), and identifying build dependencies. Approximately 16 subjects will participate in this study. If you agree to participate, you will be asked to perform a series of build maintenance-related tasks on a given software project both with and without our build maintenance tool "Formiga", as well complete a short survey afterwards. This will take approximately 90-150 minutes of your time.

Risks that you may experience from participating are considered minimal. There are no costs for participating. There are no benefits to you other than to further research.

Your responses will be treated as confidential and all reasonable efforts will be made so that no individual participant will be identified with his/her answers. Identifying information such as your name will be collected only to link data related to your activities. The research team will remove your identifying information after analyzing the data and all study results will be reported without identifying information so that no one viewing the results will ever be able to match you with your responses. Data from this study will be saved on a networked, password-protected computer in a locked room (EMS 1010) until May 2016. Only Ryan Hardt and Ethan Munson will have access to your information. However, the Institutional Review Board at UW-Milwaukee or appropriate federal agencies like the Office for Human Research Protections may review this study's records.

Your participation in this study is voluntary. You may choose not to take part in this study, or if you decide to take part, you can change your mind later and withdraw from the study. You are free to not answer any questions or withdraw at any time. Your decision will not change any present or future relationships with the University of Wisconsin Milwaukee.

If you have questions about the study or study procedures, you are free to contact the investigator at the address and phone number below. If you have questions about your rights as a study participant or complaints about your treatment as a research subject, contact the Institutional Review Board at (414) 229-3173 or irbinfo@uwm.edu.

To voluntarily agree to take part in this study, you must be 18 years of age or older. By completing the survey, you are giving your consent to voluntarily participate in this research project.

Thank you!

Ryan Hardt
Department of EECS
P.O. Box 784
Milwaukee, WI 53201
(414) 229-6479
rrhardt@uwm.edu

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: adding files

A new unit of time measurement is needed to represent a decade. You are asked to add a java source file named “Decade.java” in the org.jfree.data.time package in the “source” directory at the project root. You are also asked to identify the build targets and tasks that will be directly affected by this new file. A build task is “directly affected” by a new file if that task uses it as input in its original location.

- Add “Decade.java” to the org.jfree.data.time package in the “source” directory
- Identify the build targets and tasks that will be directly affected by this new file

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: adding files

A new unit of time measurement is needed to represent a decade. You are asked to add a java source file named “Decade.java” in the org.jfree.data.time package in the “source” directory at the project root. You are also asked to identify the build targets and tasks that will be directly affected by this new file. A build task is “directly affected” by a new file if that task uses it as input in its original location.

- Identify the build targets and tasks that will be directly affected by adding the file “Decade.java” to the org.jfree.data.time package in the “source” directory (but do not actually add the file)

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: adding files

A new JUnit test is needed for a class named “Decade.java” which resides in the org.jfree.data.time package in the “source” directory at the project root. You are asked to create the JUnit test “DecadeTest.java” in the org.jfree.data.time package in the “tests” directory at the project root. You are also asked to identify the build targets and tasks that will be directly affected by this new file. A build task is “directly affected” by a new file if that task uses it as input in its original location.

- Add the JUnit test named “DecadeTest.java” to the org.jfree.data.time package in the “tests” directory
- Identify the build targets and tasks that will be directly affected by this new file

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: adding files

A new JUnit test is needed for a class named “Decade.java” which resides in the org.jfree.data.time package in the “source” directory at the project root. You are asked to create the JUnit test “DecadeTest.java” in the org.jfree.data.time package in the “tests” directory at the project root. You are also asked to identify the build targets and tasks that will be directly affected by this new file. A build task is “directly affected” by a new file if that task uses it as input in its original location.

- Identify the build targets and tasks that will be directly affected by adding the JUnit test “DecadeTest.java” to the org.jfree.data.time package in the “tests” directory (but do not actually add the file)

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: renaming files

A new version of the pom.xml file needs to be created. In the meantime, you are asked to rename “pom.xml” (which is located at the project root) to “pom-old.xml” and update the build to account for this. The file should still be used the same way within the build. This means that if the file was previously used by some task, it is still used by that task. If the file was not previously used by some task, it is still not used by that task.

- Rename “pom.xml” (located at the project root) to “pom-old.xml”
- Update the build to account for this change

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: renaming files

A new version of the pom.xml file needs to be created. In the meantime, you are asked to rename “pom.xml” (which is located at the project root) to “pom-old.xml” and update the build to account for this. The file should still be used the same way within the build. This means that if the file was previously used by some task, it is still used by that task. If the file was not previously used by some task, it is still not used by that task.

- Update the build to account the rename of “pom.xml” (located at the project root) to “pom-old.xml” (but do not actually rename the file)

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: renaming files

A decision has been made to augment the servlet jar with its version number. Rename “servlet.jar” (located in the “lib” directory at the project root) to “servlet-2.3.1.jar”. The file should still be used the same way within the build. This means that if the file was previously used by some task, it is still used by that task. If the file was not previously used by some task, it is still not used by that task.

- Rename “servlet.jar” (located in the “lib” directory) to “servlet-2.3.1.jar”
- Update the build to account for this change

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: renaming files

A decision has been made to augment the servlet jar with its version number. Rename “servlet.jar” (located in the “lib” directory at the project root) to “servlet-2.3.1.jar”. The file should still be used the same way within the build. This means that if the file was previously used by some task, it is still used by that task. If the file was not previously used by some task, it is still not used by that task.

- Update the build to account for the rename of “servlet.jar” (located in the “lib” directory) to “servlet-2.3.1.jar” (but do not actually rename the file)

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: moving files

The application is going to add an emphasis on pie charts. You are responsible for maintaining documentation. You are asked to create a new directory at the project root named “docfiles-pie” and move the file “docfiles/PiePlotSample.png” to this newly created “docfiles-pie” directory. The file should still be used the same way within the build. This means that if the file was previously used by some task, it is still used by that task. If the file was not previously used by some task, it is still not used by that task.

- Move “PiePlotSample.png” in the “docfiles” directory to a new directory named “docifles-pie” at the project root
- Update the build to account for this change

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: moving files

The application is going to add an emphasis on pie charts. You are responsible for maintaining documentation. You are asked to create a new directory at the project root named “docfiles-pie” and move the file “docfiles/PiePlotSample.png” to this newly created “docfiles-pie” directory. The file should still be used the same way within the build. This means that if the file was previously used by some task, it is still used by that task. If the file was not previously used by some task, it is still not used by that task.

- Update the build to account for the move of “PiePlotSample.png” from the “docfiles” directory to a new directory named “docifles-pie” at the project root (but do not actually move the file or create the directory)

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: moving files

You are responsible for the French localization of the editor functionality. A decision has been made to move the property files from source/org/jfree/chart/editor to a new directory at the project root named “properties”. You are asked to create the new directory and move the file source/org/jfree/chart/editor/LocalizationBundle_fr.properties to this new “properties” directory. The file should still be used the same way within the build. This means that if the file was previously used by some task, it is still used by that task. If the file was not previously used by some task, it is still not used by that task.

- Move “LocalizationBundle_fr.properties” in the “source/org/jfree/chart/editor” directory to the newly created “properties” directory
- Update the build to account for this change

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: moving files

You are responsible for the French localization of the editor functionality. A decision has been made to move the property files from source/org/jfree/chart/editor to a new directory at the project root named “properties”. You are asked to create the new directory and move the file source/org/jfree/chart/editor/LocalizationBundle_fr.properties to this new “properties” directory. The file should still be used the same way within the build. This means that if the file was previously used by some task, it is still used by that task. If the file was not previously used by some task, it is still not used by that task.

- Update the build to account for the move of “LocalizationBundle_fr.properties” from the “source/org/jfree/chart/editor” directory to the newly created “properties” directory (but do not actually move the file or create the directory)

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: deleting files

You are told that jfreechart will no longer be licensed under lgpl. You are asked to delete the file “licence-LGPL.txt” (located at the project root) and account for this removal in the build. You are also asked to identify any targets and tasks that will be directly affected by this change. A build task is “directly affected” by a deleted file if that task previously used it as input in its original location.

- Delete the file “licence-LGPL.txt” (located at the project root)
- Update the build to account for this change
- Identify any targets and tasks that will be directly affected by this change

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: deleting files

You are told that jfreechart will no longer be licensed under lgpl. You are asked to delete the file “licence-LGPL.txt” (located at the project root) and account for this removal in the build. You are also asked to identify any targets and tasks that will be directly affected by this change. A build task is “directly affected” by a deleted file if that task previously used it as input in its original location.

- Update the build to account for the deleted file “licence-LGPL.txt”, located at the project root (but do not actually delete the file)
- Identify any targets and tasks that will be directly affected by this change

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: deleting files

You are told that the image located at source/org/jfree/chart/gorilla.jpg is no longer necessary. You are asked to delete the file and account for this removal in the build. You are also asked to identify any targets and tasks that will be directly affected by this change. A build task is "directly affected" by a deleted file if that task previously used it as input in its original location.

- Delete the file "gorilla.jpg" located in the "source/org/jfree/chart/" directory
- Update the build to account for this change
- Identify any targets and tasks that will be directly affected by this change

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build maintenance due to project refactoring
Subcategory: deleting files

You are told that the image located at source/org/jfree/chart/gorilla.jpg is no longer necessary. You are asked to delete the file and account for this removal in the build. You are also asked to identify any targets and tasks that will be directly affected by this change. A build task is "directly affected" by a deleted file if that task previously used it as input in its original location.

- Update the build to account for the deleted file "gorilla.jpg" located in the "source/org/jfree/chart/" directory (but do not actually delete the file)
- Identify any targets and tasks that will be directly affected by this change

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: dependency discovery
Subcategory: forward dependencies

You are told that the file “CombinedXYPlot.java” in the package org.jfree.experimental.chart.plot in the “experimental” directory needs to change and are asked what deliverables will need to be rebuilt. Name all of the deliverables that the file “CombinedXYPlot.java” contributes to. A file contributes to a deliverable if it is used by any task that is potentially executed in order to build that deliverable. At the bottom of this page is a list of all deliverables produced by jfreechart.

- Identify all of the deliverables that the file “CombinedXYPlot.java” in the package org.jfree.experimental.chart.plot contributes to

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

JFreeChart Deliverables

- jfreechart-1.0.16.jar
- jfreechart-1.0.16-experimental.jar
- jfreechart-1.0.16-javadocs.jar
- jfreechart-1.0.16.zip
- jfreechart-1.0.16.tar.gz
- jfreechart-1.0.16-bundle.jar

Formiga Usability Study

Category: dependency discovery
Subcategory: forward dependencies

You are told that the file “DialPlotSample.png” in the “docfiles” directory is going to change and are asked what deliverables will need to be rebuilt. Name all of the deliverables that the file “DialPlotSample.png” contributes to. A file contributes to a deliverable if it is used by any task that is potentially executed in order to build that deliverable. At the bottom of this page is a list of all deliverables produced by jfreechart.

- Identify all of the deliverables that the file “DialPlotSample.png” in the “docfiles” directory (located at the project root) contributes to

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

JFreeChart Deliverables

- jfreechart-1.0.16.jar
- jfreechart-1.0.16-experimental.jar
- jfreechart-1.0.16-javadocs.jar
- jfreechart-1.0.16.zip
- jfreechart-1.0.16.tar.gz
- jfreechart-1.0.16-bundle.jar

Formiga Usability Study

Category: dependency discovery
Subcategory: backward dependencies

You are the developer that oversees the production of the jfreechart-1.0.16-bundle.jar deliverable. The developer responsible for maintaining the JUnit tests for the gantt charts located in the package org.jfree.data.gantt in the "tests" directory wants to know if they are currently involved in the production of the jfreechart-1.0.16-bundle.jar deliverable. A file contributes to a deliverable if it is used by any task that is potentially executed in order to build that deliverable.

- Identify whether or not the JUnit tests located in the tests/org/jfree/data/gantt directory are involved in the production of the jfreechart-1.0.16-bundle.jar deliverable

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: dependency discovery
Subcategory: backward dependencies

You are the developer that oversees the production of the jfreechart-1.0.16.tar.gz deliverable. The developer responsible for maintaining the xml documents used to validate style guidelines located in the “checkstyle” directory (located at the project root) wants to know if they are currently involved in the production of the jfreechart-1.0.16.tar.gz deliverable. A file contributes to a deliverable if it is used by any task that is potentially executed in order to build that deliverable.

- Identify whether or not the xml files located in the “checkstyle” directory (located at the project root) are involved in the production of the jfreechart-1.0.16.tar.gz deliverable

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: dependency discovery
Subcategory: graph functionality

You are told that the file “jfreechart-1.0.16-experimental.jar” in the “lib” directory is mistakenly present in the deliverable “jfreechart-1.0.16-bundle.jar” and are asked to remove this dependency. Identify the target(s) and task(s) in the build that are responsible for the inclusion of /lib/jfreechart-1.0.16-experimental.jar in the jfreechart-1.0.16-bundle.jar deliverable.

- Identify the target(s) and task(s) in the build that are responsible for the inclusion of /lib/jfreechart-1.0.16-experimental.jar in the jfreechart-1.0.16-bundle.jar deliverable

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: dependency discovery
Subcategory: graph functionality

You are told that the file “jfreechart-1.0.16-swt.jar” in the “lib” directory is mistakenly present in the deliverable “jfreechart-1.0.16.tar.gz” and are asked to remove this dependency. Identify the target(s) and task(s) in the build that are responsible for the inclusion of /lib/ jfreechart-1.0.16-swt.jar in the jfreechart-1.0.16.tar.gz deliverable.

- Identify the target(s) and task(s) in the build that are responsible for the inclusion of /lib/jfreechart-1.0.16-swt.jar in the jfreechart-1.0.16.tar.gz deliverable

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build refactoring
Subcategory: renaming properties

You decide that the “jfreechart.version” property should be renamed to “version” and decide to do so.

- Rename the “jfreechart.version” property to “version”

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build refactoring
Subcategory: renaming properties

You decide that the “jcommon.name” property should be renamed to “jcommon” and decide to do so.

- Rename the “jcommon.name” property to “jcommon”

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build refactoring
Subcategory: renaming targets

You notice that the “compile” target is actually compiling source files and creating a jar file. You decide to rename the target to “compile-jar”.

- Rename the “compile” target to “compile-jar”

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build refactoring
Subcategory: renaming targets

There is some discussion about moving away from zip files for distribution. Just in case, you decide to change the name of the “zip” target to “archive”.

- Rename the “zip” target to “archive”

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build refactoring
Subcategory: removing properties

You decide that the “jfreechart.name” property is unnecessary and decide to remove it. Wherever the property is referenced, you will use the property’s current value, which is “jfreechart”.

- Remove the “jfreechart.name” property
- Replace its references with its current value, which is “jfreechart”

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build refactoring
Subcategory: removing properties

You decide that the “jcommon.version” property is unnecessary and decide to remove it. Wherever the property is referenced, you will use the property’s current value, which is “1.0.20”.

- Remove the “jcommon.version” property
- Replace its references with its current value, which is “1.0.20”

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build refactoring
Subcategory: removing targets

You are told that the build will no longer be responsible for JUnit testing and are asked to update the build as a result. You are told that the “test” target is responsible for running the tests. Remove JUnit testing from the build.

- Remove JUnit testing from the build

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Usability Study

Category: build refactoring
Subcategory: removing targets

You are told that the build will no longer be responsible for creating javadoc files and are asked to update the build as a result. You are told that the “zip-javadocs” target is responsible for generating the archive of javadoc files. Remove the javadoc functionality from the build.

- Remove javadoc functionality from the build

Please indicate when you are ready to begin the task. Please also indicate when you have completed the task.

Formiga Survey

Please answer the following questions about *how useful you feel Formiga is for the following tasks* according to the scale below. All questions are optional.

Evaluation Scale: (5) Excellent (4) Very Good (3) Good (2) Fair (1) Poor

Build Maintenance Due to Source Code Refactoring
These questions deal with Formiga's responses to updates made to a project's source code and other project resources. How would you rate Formiga's usefulness for the following tasks:

Adding files	5	4	3	2	1
Renaming files	5	4	3	2	1
Moving files	5	4	3	2	1
Deleting files	5	4	3	2	1

Dependency Discovery
These questions deal with Formiga's graphing functionality for dependency identification. How would you rate Formiga's usefulness for the following tasks:

Identifying how a file is used by the build system*:	5	4	3	2	1
Identifying files involved in building a deliverable**:	5	4	3	2	1
Identifying dependency construction locations in the build:	5	4	3	2	1

* "forward dependencies"
**"backward dependencies"

Build Refactoring
These questions deal with Formiga's build refactoring capabilities. How would you rate Formiga's usefulness for the following tasks:

Renaming properties	5	4	3	2	1
Renaming targets	5	4	3	2	1
Removing properties	5	4	3	2	1
Removing targets	5	4	3	2	1

Overall
Overall, how would you rate Formiga's usefulness?

Overall	5	4	3	2	1

Please answer the following questions about you and your level of experience with Ant *prior to this study*. All questions are optional.

If you have professional software development experience, how many years and/or months have you been working as a developer?

If you are currently a computer science student, how many semesters have you been studying computer science?

How many years and/or months have you been using Ant?

How would you describe your knowledge of Ant?

☐ None ☐ Little ☐ Moderate ☐ Experienced ☐ Expert

What did you like best about Formiga?

What did you like least about Formiga?

Are there any capabilities that you wish Formiga had?

Bibliography

[1] S. Muller and T. Fritz, "Stakeholders' information needs for artifacts and their dependencies in a real world context," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 290–299, 2013.

[2] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), pp. 141–150, ACM, 2011.

[3] S. McIntosh, "Build system maintenance," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), pp. 1167–1169, ACM, 2011.

[4] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, "The evolution of the Linux build system," *ECEASST*, vol. 8, 2007.

[5] 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.

[6] S. McIntosh, B. Adams, and A. Hassan, "The evolution of ant build systems," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pp. 42 –51, May 2010.

[7] G. Kumfert and G. Epperly, "Software in the doe: The hidden overhead of "the build"," *Lawrence Livermore National Laboratory*, February 2002.

[8] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pp. 114–123, Oct. 2007.

[9] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, (Piscataway, NJ, USA), pp. 650–660, IEEE Press, 2012.

[10] D. Spinellis, "Software builders," *Software, IEEE*, vol. 25, pp. 22 –23, May-June 2008.

[11] "Gnu make." `http://www.gnu.org/software/make/`, January 2014.

[12] "The apache ant project." `http://ant.apache.org`, January 2014.

[13] N. Serrano and I. Ciordia, "Ant: automating the process of building applications," *Software, IEEE*, vol. 21, pp. 89 – 91, Nov.-Dec. 2004.

[14] "Maven - welcome to apache maven." `http://maven.apache.org`, February 2014.

[15] "Gradle - build automation evolved." `http://www.gradle.org`, February 2014.

[16] "Groovy - home." `http://groovy.codehaus.org`, February 2014.

[17] S. Dart, "Spectrum of functionality in configuration management systems," tech. rep., 1990.

[18] D. B. Leblang and R. P. Chase, Jr., "Computer-aided software engineering in a distributed workstation environment," *SIGSOFT Softw. Eng. Notes*, vol. 9, pp. 104–112, April 1984.

[19] K. Marzullo and D. Wiebe, "Jasmine: a software system modelling facility," in *Proceedings of the second ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 2, (New York, NY, USA), pp. 121–130, ACM, 1987.

[20] A. Mahler and A. Lampen, "Shape - a software configuration management tool," in *Proceedings of the International Workshop on Software Version and Configuration Control*, (Grassau, West Germany), pp. 228–243, B. G. Teubner, January 1988.

[21] A. Rich and M. Solomon, "A logic-based approach to system modelling," in *Proceedings of the 3rd international workshop on Software configuration management*, SCM '91, (New York, NY, USA), pp. 84–93, ACM, 1991.

[22] Y.-J. Lin and S. P. Reiss, "Configuration management with logical structures," in *Proceedings of the 18th international conference on Software engineering*, ICSE '96, (Washington, DC, USA), pp. 298–307, IEEE Computer Society, 1996.

[23] A. Heydon, R. Levin, T. Mann, and Y. Yu, "The vesta approach to software configuration management," *Compaq Systems Research Center Research Report*, March 2001.

[24] M. C. Chu-Carroll, J. Wright, and D. Shields, "Supporting aggregation in fine grained software configuration management," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '02/FSE-10, (New York, NY, USA), pp. 99–108, ACM, 2002.

[25] "Ibm - rational clearcase - united states." `http://www-03.ibm.com/software/products/en/clearcase`, January 2014.

[26] L. Hochstein and Y. Jiao, "The cost of the build tax in scientific software," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pp. 384–387, 2011.

[27] J. Estublier, D. Leblang, A. v. d. Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber, "Impact of software engineering research on the practice of software configuration management," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 383–430, October 2005.

[28] J. Buffenbarger, "Adding automatic dependency processing to makefile-based build systems with amake," in *Release Engineering (RELENG), 2013 1st International Workshop on*, pp. 1–4, 2013.

[29] "Virtual ant." http://www.placidsystems.com/virtualant/, January 2014.

[30] "Vizant - ant task to visualize buildfile." http://vizant.sourceforge.net/, January 2014.

[31] "Apache ivy." http://ant.apache.org/ivy/index.html, January 2014.

[32] "Apache derby." http://db.apache.org/derby/, January 2014.

[33] "xmlgraphics.apache.org." http://xmlgraphics.apache.org, January 2014.

[34] "Hibernate. everything data. - hibernate." http://hibernate.org, January 2014.

[35] "Jung - java universal network/graph framework." http://jung.sourceforge.net, February 2014.

[36] "Freechart." http://www.jfree.org/jfreechart/, February 2014.

CURRICULUM VITAE

Ryan Hardt

Place of birth: Appleton, WI

Education

B.A., Computer Science, Carthage College, May 2004

B.A., Mathematics, Carthage College, May 2004

M.S., Computer Science, Univeristy of Wisconsin-Milwaukee, August 2008

Dissertation Title: Ant Build Maintenance with Formiga

Conference Publications (Full Length)

R. Hardt, E. V. Munson and H. Nguyen. "A Search Engine for Web Images using Text Stemming," in *Proceedings of the Fourth International Conference on Web Information Systems and Technology (WEBIST 2008)*, Funchal, Portugal, Volume 2, pages 223 - 230, May 2008.

Conference Publications (Short Paper)

R. Hardt and E. V. Munson. "Ant Build Maintenance with Formiga," in *Proceedings of the First International Conference on Release Engineering (RELENG 2013)*, at 2013 International Conference on Software Engineering (ICSE 2013), San Francisco, CA, USA, May 2013.

Course Lecturer

Introducdtion to the Internet and the World Wide Web

Introduction to Computer Science Labs

Introduction to Computer Programming

Intermediate Computer Programming

iOS Development

Graduate School Service

Graduate Student Advisory Council (2012 - 2014)

Graduate Assistant Appeals Panel (2013 - 2014)