A Graph-based Framework for Reasoning about Relationships among Software Modifications

Ripon Saha, Rui Qiu[*], Miryung Kim, Dewayne Perry
Department of Electrical and Computer Engineering
The University of Texas at Austin

ABSTRACT

During the development and maintenance of large-scale software systems, developers need to be aware of other developers' changes and the implications of how and why those changes impact their own work. We introduce CHIME, a graph-based framework for finding relevant software modifications according to various notions of relevance—dependence, interference, similarity, and co-occurance. The heart of the framework is a novel data structure called ChimeGraph that stores program elements, change facts, and their temporal, spatial, and structural dependencies across multiple versions of a program. We have implemented a prototype of CHIME and applied it to three subject systems. For 459 revisions of a Java program of size over 142KLOC, the ChimeGraph construction took just over two hours. To demonstrate the flexibility of our ChimeGraph, we have defined six types of delta relationships, i.e., useful relations among program modifications and implemented corresponding graph-traversal algorithms for these delta relationships. Our initial results indicate that ChimeGraph scales to at least medium size systems and shows promise in identifying relationship between revisions in a flexible manner.

1. INTRODUCTION

Large-scale software systems are pervasive in modern society. Software engineers typically work in a collaborative development environment to build and maintain large-scale software systems. Since software modules are heavily intertwined with one another, changes to one piece of code often have a notable impact on other pieces of code, both changed and unchanged. Therefore, software engineers need to be aware of modifications implemented by others in order to resolve merge conflicts or manage potential change impacts.

Existing awareness tools either overload developers with a large volume of irrelevant information or hardwire the relevance criteria among software modifications that one should know about. For example, the most basic awareness service such as a SVN email feature monitors a repository and automatically generates an email notification for each commit, quickly flooding developers' inboxes with a large volume of irrelevant change information. Palantír [10] and FASTDash [1] strive to mitigate information overload by only notifying developers about certain change events; however, these notifications are filtered in a predefined way, e.g., notify Alice when someone else is modifying the same file she modified. YooHoo [6] reports only external API changes that lead to build errors. Chianti [9] finds only the tests affected by the changes between two program versions and a subset of edits affecting those tests.

While there exist frameworks for version history analyses, they do not explicitly model temporal, spatial, and structural dependency relationships among the software modifications. Thus, they are not designed for identifying revision pairs meeting various types of dependencies, interference, and similarity, relationships. For example, Evolizer [5] extracts and stores software modification per revision, such as method `foo()` was deleted in revision s. But Evolizer does not store structural dependencies among those modifications, such as a deleted method `foo()` was invoked by another method `bar()` in revision t.

Our ultimate goal is to create an extensible code change analysis framework, where developers can define their own notion of relevance between software revisions, and search and monitor code changes using these customized relevance criteria. As a first step towards this goal, we have developed, implemented, and evaluated a graph-based data structure, called ChimeGraph. ChimeGraph stores change facts and their temporal, spatial, and structural dependencies. ChimeGraph is designed to be flexible and extensible—users can import the results from other tools into ChimeGraph by defining new types of edges and nodes.

We have implemented a prototype tool that extracts source code from an SVN repository, computes program differences per each revision, and stores change facts and the temporal, spatial, and structural dependencies among them.

By surveying existing literature, we have identified 18 delta relationships—relationships among software modifications that developers can use to identify relevant changes made by other developers. For example, if a developer creates method m_1 in revision s that overrides method m_2 created by an-

[*]The first two authors collaboratively implemented and evaluated the work.

other developer in revision t, we say there is a delta relationship between revision s and revision t. Such relationships could potentially help developers detect unanticipated behavior changes, or help the two developers recognize how the two revisions may impact one another. We have implemented corresponding graph reachability algorithms for six delta relationships.

We constructed ChimeGraph on three Java programs of varying sizes. For a program history of 142KLOC and 459 revisions, ChimeGraph construction took just over two hours. Running the six types of delta relationship queries on these graphs took less than two seconds. These initial results suggest that ChimeGraph scales well to at least medium size systems.

In the remainder of this paper, we describe the design of ChimeGraph in Section 2, and our implementation details and preliminary results in Section 3. We then present our future work in Section 4.

2. CHIMEGRAPH

ChimeGraph is a dependency graph that models changes to program elements (e.g., packages, classes, methods, etc.) and their inter-dependencies (e.g. containment relations, subtype relations, calling relations, etc.) across multiple revisions of a program. We represent each program element as a ChimeGraph node and each dependency between two elements as a ChimeGraph edge. To concisely represent additions and deletions of program elements in ChimeGraph, we add a lifetime $[r_b, r_e]$ to corresponding nodes. Here r_b represents the revision number where the element was added and $r_e$represents the revision number where the element was deleted. In Figure 2, `PL`, `CL`, `ML`, and `FL` represent the lifetime nodes for packages, classes, methods, and fields respectively.

In addition to defining nodes for source code elements, we have defined another type of node, *change node*, that represents modifications to any code element that ever existed in the version history. The change node contains information about the nature of the changes and the revision number in which the changes occurred. We define two different types of change nodes: *method body change* (`MC`) and *lookup change* (`LC`). While an `MC` node represents any syntactic changes to a method body, an `LC` node represents changes which affect the lookup table of a program by altering its dynamic dispatching behavior. For example, a method call could be redirected to a different method due to an additional overriding method definitions, changes to the class hierarchy, or other forms of dynamic dispatching change. Therefore, the ChimeGraph data structure represents not only syntactic changes but also behavioral changes of object-oriented programs.

We currently model four types of inter-dependencies among individual nodes: containment edges to represent spatial dependencies, reference edges (method calls, field accesses, and object instantiations), and subtyping edges (`extends` or `implements` relations in Java) to represent structural dependencies, and change order edges to represent temporal dependencies. We add a lifetime for each reference and subtyping edges to keep track of the appearances and disappearances of dependencies. For example, a call dependency edge from method m_1 to method m_2 with a lifetime [2,10] indicates that method m_2 called method m_1 at revision 2, but the call has been deleted at revision 10. Thus, from the ChimeGraph, one can easily extract the change history of a program element along with its dependency on other elements by simply traversing the edges of the corresponding node. The lifetime information also allows a *violation detection algorithm* to detect any inconsistencies (e.g. definition of a method is deleted while its call is still present) in the graph by checking the lifetimes of edges and associated source nodes.

Finally, ChimeGraph is an extensible data structure capable of accommodating finer-grained information and additional definitions of relevance beyond what we have already implemented. For example, developers may add *renaming edges* to describe how deleted nodes in one version correspond to added nodes in the next version.

A Concrete Example. We now present a simple code example comprising three revisions (see Figure 1) to illustrate how we model change facts and their inter-dependencies in actual ChimeGraph. Suppose that all elements of revision 1 are considered to be *added* with respect to revision 0 and that one package, three classes, and three method definitions were added in revision 1. To model these changes, we first create a package lifetime node `PL(1,∞, p1)` to represent that package `p1` was added, and three class lifetime nodes `CL(1,∞, p1.A)`, `CL(1,∞, p1.B)`, and `CL(1,∞, p1.C)` to represent the addition of three classes A, B, and C respectively. For all newly added nodes, a lifetime of $[1,\infty]$ indicates that these nodes were created in revision 1 and have not yet been deleted. In a similar fashion, we create all method lifetime (`ML`) nodes for each method definition and add the containment edges to reflect their spatial relationships.

Since class `B` extends class `A`, we add a subtyping edge from node `CL(1,∞, p1.A)` to `CL(1,∞, p1.B)` to indicate that super class `A` must exist before sub class `B` can exist. Similarly, since `B.bar()` is calling `A.foo()`, we add a reference edge `(1,∞,MI)` from `ML(1,∞, p1.A.foo)` to `ML(1,∞, p1.B.bar)`. The parameter `MI` stands for method invocation. The lifetime of both the reference edge and associated nodes allow us easily to detect accidental deletions of methods that lead to build errors. In revision 2, a method `B.foo` is added that overrides `A.foo()`. Because of this change, the invocation of `B.foo` inside method `B.bar` is no longer referring to `A.foo`. We model this change by adding a lookup change node `LC(B.foo,A.foo)` and connect it from `ML(2,∞,p1.B.foo)` node using a change order edge. We continue in this way to populate the remainder of the ChimeGraph. To model deletions, we update the lifetime of the corresponding node instead of deleting the node entirely. In the example code base, `B.bar()` is deleted in revision 3. We can easily obtain this information from the ChimeGraph by observing the lifetime of that node.

3. IMPLEMENTATION AND EVALUATION

Our current prototype is implemented as an Eclipse plugin. CHIME extracts program versions from a target SVN repository using SVNKit[1] and uses an Eclipse JDT abstract syntax tree analysis to analyze each program version. For each consecutive version pair, it compares each file's AST us-

[1]http://svnkit.com/

Revision 1

```
package p1;
Class A {
  void foo(){}
}
Class B extends A {
  void bar() {
    foo();
  }
}
Class C {
  B b;
  void baz() {
    b = new B();
    b.bar();
  }
}
```

Revision 2

```
package p1;
Class A {
  void foo(){}
  void boo(){}
}
Class B extends A {
  void foo() {}
  void bar() {
    foo();
  }
}
Class C {
  B b;
  void baz(int a) {
    b = new B();
    b.bar();
  }
}
```

Revision 3

```
package p1;
Class A {
  void foo() {
    boo();
  }
  void boo(){}
}
Class B extends A {
  void foo() {};
  // B.bar() was deleted
}
Class C {
  B b;
  void baz(int a) {
    b = new B();
    b.bar();
  }
}
```

Figure 1: ChimeGraph Code Example

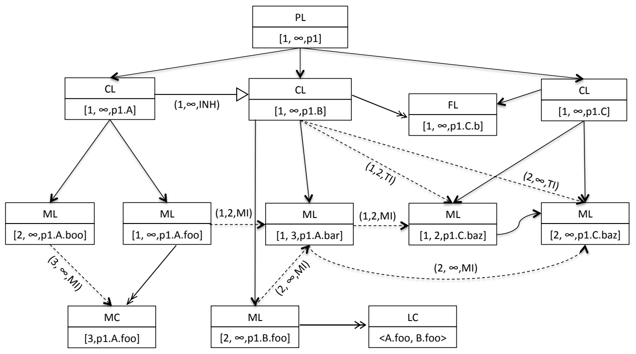

Figure 2: ChimeGraph Data Structure Example

Table 1: Subject Systems

Name	LOC	#Rev.	#Nodes	#Edges
ChimeGraph	5492	52	2138	3582
JUnit	17,509	650	11481	19368
Columba	142,432	459	28519	63209

ing ChangeDistiller's [4] tree differencing algorithm. It then builds and updates ChimeGraph incrementally based on program differencing results. Furthermore, to resolve types as much as possible for incomplete, uncompilable programs, it uses the partial program analysis (PPA) [3]. It stores ChimeGraph in an XML file for future use. We implement graph traversal algorithms to encode each of the six delta relationships represented in Table 2.

Evaluation. To evaluate the scalability of ChimeGraph, we have used three Java projects: *ChimeGraph*, *JUnit*, and *Columba* (see Table 1). Our evaluation addresses the following two research questions.

RQ1: Can CHIME serve as a basis for identifying various delta relationships surveyed in the literature? We have first compiled various temporal, spatial, and structural dependency relationships among software modifications mentioned or implemented in the literature [7]. Out of 18 delta relationships identified by the survey, we have implemented 6 of them because these relationships are currently supported by the granularity of change facts and dependencies in our graph. (We plan to make ChimeGraph extensible and to demonstrate that the remaining 12 delta relationships can be supported based on ChimeGraph.) Because ChimeGraph represents spatial, temporal, and structural dependencies among change facts and encodes the lifetime of individual code elements explicitly, it is fairly simple to encode those delta relationships as a graph traversal algorithm. For example, to detect the delta relationship #1 in Table 2, we can simply traverse all method invocation edges and check the lifetime of the associated nodes and the edges between them to identify a set of revision pairs where one deletes code elements but the corresponding references to them are not deleted accordingly. Timely identification of these delta relationships can help identify potential build errors caused by other developers. Table 3 reports number of relevant revision pairs that we have identified for each delta relationship. We also calculate the percentage of identified delta relationships among all possible revision pairs.

Table 2: Various Delta Relationship Definitions

No.	Relationships between two revisions s and t
DR1	Revision s introduces a call dependency for a method that is deleted in revision t while the call still exists [6].
DR2	Revision s introduces a call dependency for a method whose visibility decreased by t. For example, a public method is changed to private[6].
DR3	Revision s overrides a method which is created in revision t [2].
DR4	Both revision s and revision t modify the same method [10].
DR5	Revision s introduces a call dependency for a method whose implementation changed in revision t [6].
DR6	Revision s and revision t insert the same structural dependencies [8]. For example, both revisions introduce a call dependency for the same method.

RQ2: Can CHIME handle version histories of real world projects? To assess scalability of ChimeGraph, we measured the time taken for constructing a graph and running graph-traversal algorithms for individual delta relationships. We used a 3.33 GHz Intel Core2Duo iMac computer

Table 3: Number of Relevant Revision Pairs

Name	DR1	DR2	DR3	DR4	DR5	DR6
ChimeGraph	15	0	57	121	178	167
	1.1%	0%	4.2%	9.1%	13.4%	12.6%
JUnit	252	24	468	660	956	671
	0.1%	0.01%	0.2%	0.3%	0.4%	0.3%
Columba	368	4	657	1695	2309	2341
	0.3%	0.003%	0.6%	1.6%	2.2%	2.2%

Table 4: Performance

Name	Update Time	Query Time	XML Size
ChimeGraph	4m 25s	0.153s	3.2MB
JUnit	58m 21s	0.673s	19.8MB
Columba	129m 50s	1.393s	53.8MB

and allocated 2GB of memory to run the program. It took less than an hour to handle 650 revisions of *JUnit*, 4.5 min to handle 52 revisions of our own *ChimeGraph* version history, and 2 hours and 10 minutes to handle 459 revisions of *Columba* (see Table 4). The running time for identifying all six types of delta relationships took less than two seconds in all three subjects. On the basis of these results, our graph construction and traversal shows promises in handling version histories of medium size, real world Java projects.

Comparison with Evolizer. To the best of our knowledge, existing version history analysis frameworks model only individual change facts per each revision but are not suitable for the relationships among software revisions. For example, Evolizer stores individual syntactic change facts per revision in the form of a relational database. Though comparing Evolizer and CHIME is like comparing apples and oranges, to demonstrate the suitability of ChimeGraph for reasoning about delta relationships, we tried to encode the six types of delta relationships as MySQL queries to be run on the Evolizer database. For each delta relationship query, we found the required data are stored in multiple tables, thus requiring expensive `JOIN` operations to create the needed relationship. For example, all the queries need to access at least four tables: `SourceCodeEntity` to get information about program elements, `SourceCodeChange` for the changed elements, `StructureEntityVersion` to get the version when the elements were changed, and `Transaction_Revision` for converting version numbers to revision numbers. In contrast, it is easy to identify the six delta relationships among revisions in ChimeGraph, because ChimeGraph explicitly represents the lifetime information of program elements and stores the temporal, spatial, and structural dependencies using various types of edges. For example, to identify all pairs of DR4, we can traverse each method lifetime node (`ML`) and find all outgoing edges with a destination being a method body change node (`MC`) - that is, there will be a set of `MC` nodes that are the destinations of a `ML` (see Figure 2). The output of DR4 would be all possible pairs of this set. Assume that method `foo()` is changed in revisions 2,3, and 4. There are then three `MC` nodes connected from the `ML` node of `foo()` and the result of DR4 is (2,3), (2,4), and (3,4).

4. FUTURE WORK

Our ChimeGraph prototype models only code elements and change facts at or above the granularity of methods and fields and a limited set of structural, temporal, spatial dependencies among them. Due to this limitation, we were able to encode only 6 out of 18 delta relationships we have currently identified. Our ultimate vision is to design an extensible framework where developers can define customized *delta relationships* in a flexible manner and use these relationships to search and monitor program changes relevant to their own modifications. To achieve this vision, our data representation should be *extensible* to accommodate new types of change facts or inter-dependencies among them. For example, to keep track of the lifetime of code despite renamings or refactorings, developers may want declare refactoring (or mapping) edges between deleted methods in the old version and the corresponding added methods in the next version. We plan to provide a porting functionality in our framework so that developers can add new types of nodes and edges in ChimeGraph to import the results of existing tools, such as an API matching tool for creation of refactoring edges [8].

Furthermore, our framework should not require developers to hand code graph-traversal algorithms to identify software revisions of interest based on these new types of edges and nodes. To relieve developers' burden in encoding various delta relationships in terms of traversal algorithms on the ChimeGraph, we plan to convert the edges and nodes into a logic fact data base representation and provide a logic query language for developers to declare delta relationships of interest in terms of a logical query. We also plan to provide a graphical user interface, where developers can define new queries (or refine existing ones) to investigate the relationships among software revisions.

5. REFERENCES

[1] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proc. of CHI '07*, pages 1313–1322, 2007.

[2] O. C. Chesley, X. Ren, and B. G. Ryder. Crisp: A debugging tool for java programs. In *Proc. of ICSM '05*, pages 401–410, 2005.

[3] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. In *Proc. of OOPSLA '08*, pages 313–328, 2008.

[4] B. Fluri, M. Wursch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. on Software Engineering*, 33(11):725 –743, 2007.

[5] H. C. Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. *IEEE Trans. on Software Engineering*, 26(1):26–33, Jan. 2009.

[6] R. Holmes and R. J. Walker. Customized awareness: recommending relevant external change events. In *Proc. of ICSE '10*, pages 465–474, 2010.

[7] M. Kim. An exploratory study of awareness interests about software modifications. In *Proc. of CHASE '11*, CHASE '11, pages 80–83, 2011.

[8] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Proc. of ICSE '07*, pages 333–343, 2007.

[9] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *Proc. of OOPSLA '04*, pages 432–448, 2004.

[10] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: raising awareness among configuration management workspaces. In *Proc. of ICSE '03*, pages 444–454, 2003.