

RESEARCH STATEMENT

MIRYUNG KIM

I am interested in **software engineering**, more specifically in **software evolution**. My research focuses on answering fundamental questions about software evolution: Why is the cost of evolving software so high? How do software systems actually evolve? Do current beliefs about good development practices in fact aid software evolution?

I believe in comprehensive approaches toward answering these questions. I gather hypotheses about how programmers build and maintain software through **ethnographic studies**. I then test these hypotheses by **mining historical evidence** from source code repositories. Based on these empirical understandings, I build **software engineering tools and infrastructures** to help programmers cope with difficulties in evolving software. A common theme of my research has been to automatically discover and formally represent explicit structures that approximate how programmers think about software changes. I will first describe the tool and infrastructure that I have built and then describe the empirical studies that motivated them.

PROGRAM CHANGE STRUCTURES

Programmers constantly update software to provide new features to customers and fix defects in existing software. It is often difficult for programmers to implement or understand software changes especially when the changes involve modifications across many different files. For example, modifying a locking protocol, even though it is a conceptually simple modification to software, involves crosscutting changes at a code level. My Ph.D research addresses the following key questions about software change: (1) what is a **concise and explicit representation** of a program change? and (2) how do we **automatically extract** the differences between two program versions into such representations? Answering these questions is a key step towards understanding software evolution better, reducing developer efforts in modifying software, and developing efficient analysis techniques for ever-changing software.

A high-level software change is often systematic, consisting of many homogeneous transformations at a code level. By identifying such homogeneity, my approach discovers structures that explain how individual code changes are related to one another. To explicitly capture such structures, I developed novel **rule-based change representations** (first order logic rules) and built **program differencing tools** using my rule inference algorithms. In my Ph.D thesis, I built two different rule-based representations. I invented the first type of rules to build a code matching infrastructure and then extended them to represent structural differences concisely.

Code Matching Infrastructure for Mining Software Repositories. Matching corresponding code elements between program versions is a fundamental building block for mining program change patterns from source code repositories. To match corresponding Java methods between two versions, I invented a rule-based change representation that groups similar API-level refactorings. For example, when the same input parameter is added to a set of related methods, this change is represented as a single rule not as multiple *add parameter* refactorings. I built an inference algorithm that assesses each rule's likelihood to find method-level matches with high precision and recall (Kim et al., ICSE 2007). A comparative evaluation shows that my tool has 6-26% higher recall and roughly the same precision, yet produces 22-77% more concise results than existing code matching tools.

Discovering and Representing Logical Structure in Code Changes. While *diff* represents program changes as line-level textual differences, programmers often think about program changes in terms of structural differences: Which code elements changed? How does the change affect structural dependencies such as call dependencies, data accesses, and subtyping relationships? To bridge this gap, I developed a *logical structural diff (lsdiff)* that computes structural differences as logic rules and facts (Kim et al., 2007). Each rule represents a set of transformations that share similar structural characteristics. For example, when a programmer replaces all calls to `foo` with `bar`, *lsdiff* represents this as a single rule regardless of how many lines were modified in a program. In a study of three software projects' history, when *diff* output consists of 997 lines of change scattered across 16 files on average, *lsdiff* summarizes structural differences using only 7 rules and 27 facts on average. *Lsdiff's* ability to explicitly capture high-level systematic changes—crosscutting concerns, refactorings, consistent updates of similar code fragments—is a key step towards building software engineering tools that leverage the semantics of software changes.

EMPIRICAL ANALYSIS OF SOFTWARE EVOLUTION

Do current beliefs about good software development practices in fact aid software evolution? It has been long believed that duplicated code fragments indicate poor software quality and factoring out the commonality among them improves software quality; thus, previous studies focused on measuring the percentage of code clones and interpreted a large (or increasing) number as an indicator for poor quality. On the other hand, I investigated how and why duplicated code is actually created and maintained using two empirical analyses. I used an edit capture and replay approach to gather insights into copy and paste programming practices. To extend this type of change-centric analysis to programs without edit logs, I developed a clone genealogy analysis that tracks individual clones over multiple versions. By focusing on how code clones actually evolve, I found that clones are not inherently bad and that we need better support for managing clones.

An Ethnographic Study of Copy and Paste Programming Practices. To unobtrusively observe copy and paste programming patterns, I developed an Eclipse IDE logger that records edit operations and built a replayer. In addition to analyzing edits, I conducted semi-structured interviews to understand associated programmer intentions (Kim et al., ISESE 2004). I found that skilled programmers often intentionally create code clones to reuse logic that cannot be abstracted due to programming language limitations (about four times per hour). They also manage clones for some period of time before they discover a common abstraction for the clones.

Investigating Myth about Code Clones. To validate my findings in a broader context, I developed a clone genealogy extractor that automatically reconstructs the history of similar code fragments from source code repositories. My study of clone genealogies indicates that refactoring (merging code clones) is not always beneficial or even applicable to many clones for several reasons (Kim et al., ESEC/FSE 2005): First, only 36-38% of clones actually change similarly with other clones in the same group. Second, 49-64% of clones cannot be easily removed using standard refactoring techniques. Third, very surprisingly, many clones (48-72%) disappear in a very short amount of a time (within an average of eight check-ins out of over at least 160 check-ins), and 26-34% of them are no longer considered as code clones because they changed differently from other clones in the same group. Refactoring of such short-lived clones may not be necessary and can be counterproductive if a programmer has to undo the refactoring. These findings have contributed to diverting research focus from automatic clone detection to clone management support.

<http://www.cs.washington.edu/homes/miryung>

These two empirical analyses motivated my work on program change structures as described earlier. I developed a code matching infrastructure to enable richer longitudinal software evolution analyses. Based on the insight that simple software changes can involve redundant transformations at a code level, I built *lsdiff* that encapsulates such changes as rules.

FUTURE RESEARCH INTERESTS

My vision is to use the semantics and structure of software change to better understand software evolution and reduce developer efforts in modifying software. In addition, I am interested in leveraging historical artifacts to help programmers make better decisions and avoid redundancies in software development.

Exploiting Change Semantics and Structures. I plan to use change-rules as a foundation to approach problems in release planning, regression testing, and cost estimation. In particular, I am interested in the following research questions: By reasoning about dependencies among program deltas, can we assist software engineers in deciding which logical changes should be shipped together and which cannot be shipped together? How can we better select and prioritize regression tests by exploiting the homogeneity and heterogeneity found in program changes? Can we better estimate software change cost using the semantic structure of past similar changes?

Past empirical studies of software evolution have primarily focused on quantitative and statistical analyses of a program over multiple versions, largely ignoring the structure and semantics of software changes between versions. How do such studies compare with qualitative studies of software evolution using inferred rules? I believe that the homogeneity and heterogeneity found in program changes may shed light on the problem of assessing software quality.

Beyond Source Code Level Change-Rules. I will investigate software changes from various angles such as an abstraction level, time, behavior, and type of software artifact. (1) My empirical studies of rules indicate that there is a need for even higher-order representations beyond rules; for example, complex refactorings and design pattern changes are often described as a heterogeneous collection of related rules. I intend to build a higher-order representation and a clustering algorithm that discovers sets of related rules. (2) The notion of a program delta must be extended from “between two versions” to “across a series of versions” as programmers often complete a single logical change in phases or submit several unrelated changes in a single check-in. (3) I would like to study differences in run-time behavior caused by a program change. While we have a common vocabulary for describing different types of source code changes, such as ‘refactoring’ and ‘crosscutting changes,’ I see a lack of common vocabulary for run-time behavior changes. How can we categorize and describe run-time behavior changes? Is there also structure in behavioral differences? Can we extend existing run-time behavior capturing techniques such as dynamic invariants or path spectra? (4) I would also like to investigate changes in requirement models and architecture diagrams. Automatically checking consistency between different abstractions of software changes may bridge the gap between how software architects plan changes and how programmers implement source-level changes.

Actively Leveraging Historical Information. Explicitly capturing the semantics of a program delta will not only help programmers in their daily tasks but also enable software engineering research to leverage historical information. My goal is to avoid redundancies in re-analyzing and re-testing changed programs. The envisioned analysis technique will be more efficient than traditional program analyses by leveraging two types of information, which are not actively used in software engineering research: (1) history-based approximation of a program’s behavior and (2) program delta semantics. For example, suppose that there is 90% confidence that a

variable `foo` never pointed to `bar` in the past revisions and the code modification does not involve a direct assignment to `foo`; when the time is limited for a whole program analysis, my analysis algorithm will use historical knowledge about aliasing relations and delta semantics to produce an approximated result.

History can help us learn which decisions are good and which decisions are bad. I see tremendous potential in using change history to help programmers make better decisions. First, inspired by my clone genealogy analysis, I plan to build a software economics model and a refactoring reminder that can suggest to programmers when to refactor duplicated code to maximize their return on refactoring investment. Second, I envision a tool that suggests how to redesign software by identifying design decisions that were intended to be encapsulated but were later inadvertently exposed to other modules.

Coping with Accidental Complexity in Software Evolution. There are many redundancies in software development: redundant code edits, redundant bug reports and subsequent efforts of triaging them, redundant efforts in deployment and configuration, etc. I believe that such redundancies are exacerbated in collaborative software development, where one developer's effort is not captured and reused by other developers. I would like to study in which task contexts redundancies occur frequently, which types of redundancies are inherent, and which types of redundancies can be avoided. Based on these studies, I will build software engineering tools that actively capture developer efforts, recognize their redundancies, and save the reusable efforts for later uses.

SELECTED REFERENCES

[**Kim et al., 2007**] Miryung Kim, Jonathan Beall, and David Notkin, *Discovering and representing logical structure in code changes*, UW-CSE-TR 07-09-01, 10 pages, September 2007 (In preparation for submission).

[**Kim et al., ICSE 2007**] Miryung Kim, David Notkin, and Dan Grossman, *Automatic inference of structural changes for matching across program versions*, ICSE '07: Proceedings of the 29th International Conference on Software Engineering (Washington, DC, USA), IEEE Computer Society, 2007, pp. 333–343.

[**Kim et al., ESEC/FSE 2005**] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy, *An empirical study of code clone genealogies*, ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (New York, NY, USA), ACM, 2005, pp. 187–196.

[**Kim et al., ISESE 2004**] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin, *An ethnographic study of copy and paste programming practices in OOP*, ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering (Washington, DC, USA), IEEE Computer Society, 2004, pp. 83–92.