Preview of Award 1213091 - Annual Project Report

Cover

Federal Agency and Organization Element to Which Report is Submitted:	4900
Federal Grant or Other Identifying Number Assigned by Agency:	1213091
Project Title:	SHF: Large: Collaborative Research: Science and Tools for Software Evolution
PD/PI Name:	Daniel Dig, Principal Investigator Brian P Bailey, Co-Principal Investigator Ralph E Johnson, Co-Principal Investigator Darko Marinov, Co-Principal Investigator
Submitting Official (if other than PD\PI):	N/A
Submission Date:	N/A
Recipient Organization:	University of Illinois at Urbana-Champaign
Project/Grant Period:	07/01/2012 - 06/30/2016
Reporting Period:	07/01/2012 - 06/30/2013
Signature of Submitting Official (signature shall be submitted in accordance with agency specific instructions)	N/A

Accomplishments

*** What are the major goals of the project?**

We propose to create a change-oriented programming environment (COPE), with its associated science and tools for software evolution. Specifically, we have five major goals:

1. **Understand transformations**. We will study how programmers perceive, recall, and communicate change in complex code. We will study how programmers understand changes and why, when, and where programmers do (not) apply them.

2. **Automate transformations**. Writing and scripting transformations today is hard, e.g., implementing a seemingly simple transformation such as Rename Method refactoring requires writing hundreds of lines of code. We must make this simpler. COPE will provide better ways for average programmers to quickly script their own transformations and for experts to implement sophisticated transformations.

3. **Compose, manipulate, and visualize transformations**. Understanding properties of transformations— e.g., commutativity, dependencies, parameterization—is central to transformation's analysis and reuse. Programmers should be able to take a sequence of transformations representing an optimization or feature or bug fix, and reapply it to different versions of a program or even completely different programs. We will develop novel approaches to manipulate transformations as well as visualize their effects.

4. **Archive and retrieve transformations**. When programs are represented as sequences of transformations, COPE must subsume and generalize the activities of version control. Instead of dealing with text edits, COPE will archive, retrieve, visualize, and merge transformations.

5. **Infer transformations**. We will develop an infrastructure within COPE to infer higher-level changes (e.g., bug fix,

security improvement) from lower-level changes, especially code edits. This will allow us to capture high-level program transformations even when the programmer did not specify them.

This is a joint project with the Department of Computer Science at the University of Texas, Austin, where Prof Don Batory leads the collaborators. We have held regular weekly teleconference meetings between the two institutions. One of the PIs from Illinois (Prof Darko Marinov) has spent one semester of his sabbatical at UT Austin, visiting Prof Batory. In addition, Prof Batory is serving on the PhD committee for one of the students at Illinois (Mohsen Vakillian) whose work is very relevant to the COPE project.

*** What was accomplished under these goals (you must provide information for at least one of the 4 categories below)?**

Major Activities:

In this first year, we have made important progress on all of the foundational components of COPE. We were able to:

- (i) develop an infrastructure to collect code evolution events such as text edits (see our ECOOP'12 paper),

- (ii) infer higher-level, known classes of transformations (i.e., refactorings) from text edits (see our ECOOP'13a paper) as well as infer previously unknown, but frequent transformations (submitted paper to FSE'13),

- (iii) develop a novel way to automatically test refactoring objects (see our ECOOP'13b paper)

- (iv) understand how Java programmers (mis)use concurrent collections. We supported automated transformations to correct erroneous, manual transformation (see our ICST'13 paper),

- (v) developed a novel interactive visualization, where the key innovation is to use principles of media skimming to the replay of software change (we will submit paper to a conference, once we conduct empirical evaluation)

Specific Objectives:

In order to accomplish our vision where code changes become first-class citizens, we leveraged the capabilities of an Integrated Development Environment (IDE) to capture code changes online. We developed a tool, CodingTracker, an Eclipse plug-in that unintrusively collects the fine-grained data about code evolution of Java programs. In particular, CodingTracker records every code edit performed by a developer. It also records many other developer actions, for example, invocations of automated refactorings, tests and application runs, interactions with VCS, etc. The collected data is so precise that it enables us to reproduce the state of the underlying code at any point in time. To represent the raw code edits collected by CodingTracker uniformly and consistently, we implemented an algorithm that infers changes as Abstract Syntax Tree (AST) node operations.

Using the higher-level of abstractions of changes to AST node operations, we designed and implemented a novel refactoring inference algorithm that analyzes code changes continuously. Currently, our algorithm infers ten kinds of refactorings performed either manually or automatically, but it can be easily extended to handle other refactorings as well. The inferred ten kinds of refactorings were previously reported as the most popular among automated refactorings. The inferred refactorings range from API-level refactorings (e.g., Rename Class), to partially local (e.g., Extract Method), to completely local refactorings (e.g., Extract Local Variable). We think the inferred refactorings are representative since they are both popular and cover a wide range of common refactorings that operate on different scope levels.

In order to infer previously unknown frequent code change patterns from a continuous sequence of AST node changes, we designed and implemented an algorithm using data mining techniques such as frequent itemset mining. There are unique challenges posed by our problem domain of program transformations, which render previous off-the- shelf data mining techniques inapplicable. First, for program transformations, we need to mine a contiguous sequence of code changes that are ordered by their timestamps, without any a-priori knowledge of where the boundaries between patterns of transformations are. Second, unlike the standard frequent itemset mining, when mining frequent code change patterns, a high level program transformation corresponding to a given pattern may contain several instances of the same kind of code change. For example, Rename Local Variable refactoring involves the same change for all references of the renamed variable. Consequently, in our mining problem, a transaction may contain multiple instances of the same item kind, thus forming itembags rather than itemsets.

When program transformations become first-class citizens, we must be able to ensure their correctness. We have taken a testing approach to correctness. Testing refactoring engines is a challenging problem that has gained recent attention in research. Several techniques were proposed to automate generation of programs used as test inputs and to help developers in inspecting test failures. However, these techniques can require substantial effort for writing test generators or finding unique bugs, and do not provide an estimate of how reliable refactoring engines are for refactoring tasks on real software projects. In this line of work, we evaluate an end-to-end approach for testing refactoring engines and estimating their reliability by (1) systematically applying refactorings at a large number of places in well-known, open-source projects and collecting failures during refactoring or while trying to compile the
refactored projects, (2) clustering failures into a small, manageable number of failure groups, and (3) inspecting failures to identify non-duplicate bugs.

To help developers understand how code has changed, we developed a novel interactive visualization tool called CodeSkimmer. The key innovation in the tool is that it applies principles of media skimming to the replay of software change. An analogy is how a person skims the pages of a physical book looking for an item (page, drawing, comment) of interest. The person skims rapidly at first and slows as s/he nears the item of interest. CodeSkimmer realizes a similar technique. The programmer can specify "interest" in changes to the current code using several filters. These filters include code-centric (all changes made to a selected region of the code), time-centric (all changes made to the code within a certain period of time), and person-centric (all changes made by a particular person) filters. CodeSkimmer re-plays the recorded changes rapidly when they are not of interest, otherwise, the tool slows playback and visually enhances the code when the changes are of interest. To enable this effect, we developed a new algorithm for determining whether a region of selected code was affected by changes in previous time steps.

We have also studied how developers perform manual changes when they use Java concurrent collections. Despite the fact that concurrent collection are widely used in practice, programmers can misuse these concurrent collections when composing two operations where a check on the collection (such as non-emptiness) precedes an action (such as removing an entry). Unless the whole composition is atomic, the program contains an atomicity violation bug. We conducted the first empirical study of CHECK-THEN-ACT idioms of Java concurrent collections in a large corpus of open-source applications. We catalog nine commonly misused CHECK-THEN-ACT

idioms and show the correct usage. We quantitatively and qualitatively analyze 28 widely-used open source Java projects that use Java concurrency collections – comprising 6.4M lines of code. We classify the commonly used idioms, the ones that are the most error-prone, and the evolution of the programs with respect to misused idioms. We implemented a tool, CTADETECTOR, to detect and correct misused CHECK-THEN-ACT idioms.

Significant Results:

In order to evaluate the usefullness of the toolset, we deployed and collected data from real code development. For example, CodingTracker collected data from 24 developers: 1,652 hours of development, 23,002 committed files, and 314,085 testcase runs. Using this rich data, we were able to shed light into the practice of software changes. We were able to answer diverse questions, such as: How much code evolution data is not stored in Version-Control-Systems (VCS)? How much do developers intersperse refactorings and edits in the same commit? How frequently do developers fix failing tests by changing the test itself? How many changes are committed to VCS without being tested? What is the temporal and spacial locality of changes?

We implemented and applied the refactoring inference algorithm to the code evolution data collected from 23 developers working in their natural environment for 1,520 hours. Using a corpus of 5,371 refactorings, we reveal several new facts about manual and automated refactorings. For example, more than half of the refactorings were performed manually. The popularity of automated and manual refactorings differs. More than one third of the refactorings performed by developers are clustered in time. On average, 30% of the performed refactorings do not reach the Version Control System.

We also evaluated our frequent change pattern algorithm on the same dataset and showed that it is effective, useful, and scales to big amounts of data. We analyzed some of the mined code change patterns and discovered ten popular kinds of high level program transformations.

We employed our test-generation approach on the Eclipse refactoring engines for Java and C, and we found and reported 77 new bugs for Java and 43 for C. Despite the seemingly large numbers of bugs, we found these refactoring engines to be relatively reliable, with only 1.4% of refactoring tasks failing for Java and 7.5% for C.

Using our last tool, CTADETECTOR, we found 282 buggy instances of erroneously performed transformations in widely used, mature open-source applications. We reported 155 to the developers, who examined 90 of them. The developers confirmed 60 as new bugs and accepted our patch. This shows that CHECK-THENACT idioms are commonly misused in practice, and correcting them is important.

Key outcomes or Other achievements:

Check-then-Act Misuse of Java Concurrent Collections (**best paper award**)
Yu Lin, Danny Dig
Proceedings of **ICST'13**.
Acceptance ratio: 25% (38/152)

A Comparative Study of Manual and Automated Refactoring
Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph Johnson, Danny Dig
Proceedings of **ECOOP'13.**
Acceptance ratio: 25% (29/116)

Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?
Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph Johnson, Danny Dig
Proceedings of **ECOOP'12**
Acceptance ratio: 21% (30/140)

Mining Continuous Code Changes to Detect Frequent Program Transformations

Stas Negara, Mihai Codoban, Danny Dig, Ralph Johnson. Under Review at **FSE'13**

*** What opportunities for training and professional development has the project provided?**

Carrying out the research activities provided professional training to six PhD students at Illinois, and one PhD student at Texas. We expect that once we have a more complete version of COPE running, we will use it for the undergrad Software Engineering classes at both Illinois and Texas.

*** How have the results been disseminated to communities of interest?**

We have published the results in top conferences in Software Engineering. We have also released all tools as free, open-source, and have advertised them at professional venues, such as conferences, visits to other universities (e.g., CMU, Oregon State U, North Carolina State U).

*** What do you plan to do during the next reporting period to accomplish the goals?**

Next steps are to conduct empirical evaluations of how well CodeSkimmer helps programmers understand changes to code relative to version control systems and real-time playback absent the use of media skimming techniques.

We will start working on deploying our transformation-centric approach on existing version control systems like Git and SVN. Rather than implementing a brand new version control system (with the danger of never being adopted by practitioners) we will enhance the capabilities of existing version control systems, providing orthogonal views of changes at different levels of abstraction.

We will consolidate the inference area, with the aim of detecting other classes of transformations, the first target are bug fixes. For this, we will use a hybrid of static/dynamic analysis, as well as machine learning techniques.

We will also conduct interviews with programmers at software companies to find out how programmers perceive, recall, and communicate changes.

Supporting Files

Filename	Description	Uploaded By	Uploaded On
checkThenAct.pdf	ICST'13 paper describing changes related to concurrent collections in Java, as well as the CTADetector tool	Daniel Dig	04/25/2013
ECOOP13-NegaraETAL-Inferring.pdf	ECOOP'13 paper describing the inference of refactorings from low-level text edits/ AST node operations	Daniel Dig	04/25/2013
GligoricETAL13RTR.pdf	ECOOP'13b paper describing the automatic generation of test inputs for refactoring engines	Daniel Dig	04/29/2013

Products

Journals

Books

Book Chapters

Thesis/Dissertations

Conference Papers and Presentations

Yu Lin and Danny Dig (3/18/13). *CHECK-THEN-ACT Misuse of Java Concurrent Collections*. International Conference on Software Testing and Verification (ICST'13). Luxemburg.

Status = PUBLISHED; Acknowledgement of Federal Support = Yes

Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig (7/3/13). *A Comparative Study of Manual and Automated Refactorings*. European Conference on Object-Oriented Programming (ECOOP'13). Montpellier, France.

Status = ACCEPTED; Acknowledgement of Federal Support = Yes

Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph Johnson, Danny Dig (6/15/12). *Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?*. European Conference on Object-Oriented Programming (ECOOP'12). Beijing, China.

Status = PUBLISHED; Acknowledgement of Federal Support = Yes

Stas Negara, Mihai Codoban, Danny Dig, Ralph E. Johnson (8/21/13). *Mining Continuous Code Changes to Detect Frequent Program Transformations*. Foundations of Software Engineering (FSE'13). St Petersburg, Russia.

Status = UNDER_REVIEW; Acknowledgement of Federal Support = Yes

Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, Darko Marinov (7/3/13). *Systematic Testing of Refactoring Engines on Real Software Projects*. European Conference on Object-Oriented Programming (ECOOP'13). Montpellier, France.

Status = ACCEPTED; Acknowledgement of Federal Support = Yes

Other Publications

Technologies or Techniques

- novel algorithms/tools for inferring (i) refactorings and (ii) frequent transformations

- novel algorithms/tools for detecting and fixing erroneous usage of concurrent collections

- novel ways to generate test inputs for validating refactoring tools

- novel algorithms for visualizing code changes

Patents

Nothing to report.

Inventions

Nothing to report.

Licenses

Nothing to report.

Websites

Title: CodingTracker Website

URL: http://codingtracker.web.engr.illinois.edu

Description: contains CodingTracker, along with the two other tools for inferring refactorings and frequent transformations

Title: CTADetector webpage

URL: http://mir.cs.illinois.edu/~yulin2/CTADetector/

Description: Contains the tool for detecting/fixing erroneous usage of concurrent collections. Also lists the bugs that we found/reported in the open-source projects

Title: Systematic Testing of Refactoring Engines on Real Software Projects

URL: http://mir.cs.illinois.edu/rtr/

Description: Lists the newly discovered bugs in the Eclipse refactoring engine

Other Products

Product Type: Software or Netware

Description: CodingTracker, CTADetector, CodeSkimmer are distributed freely, under open-source licenses (Eclipse Public License)

Other:

Participants

Research Experience for Undergraduates (REU) funding

What individuals have worked on the project?

Name	Most Senior Project Role	Nearest Person Month Worked
Stanislav Negara	Graduate Student (research assistant)	12
Yu Lin	Graduate Student (research assistant)	6
Daniel Dig	PD/PI	3
Brian P Bailey	Co PD/PI	0
Ralph E Johnson	Co PD/PI	0
Mihai Codoban	Graduate Student (research assistant)	1
Mohsen Vakilian	Graduate Student (research assistant)	1
Connor Simmons	Graduate Student (research assistant)	5
Darko Marinov	Co PD/PI	1
Milos Gligoric	Graduate Student (research assistant)	3

What other organizations have been involved as partners?

Name	Location

University of Texas	Austin

Have other collaborators or contacts been involved? N

Impacts

What is the impact on the development of the principal discipline(s) of the project?

the CTADetector tool found hundreds of new bugs in real-world, mature open-source applications. We have reported 90 of these bugs, and at least 60 of them have been fixed as a result. The empirical findings educate Java developers about common mistakes and pitfalls when using concurrent collections. The approach for testing refactorings has found 120 new bugs in production quality refactoring tools from the Eclipse IDE. The developers of Eclipse have started to fix some of these bugs.

The inference and visualization techniques are the basis of elevating software changes at higher levels of abstraction. Besides the impact on research and development of software engineerings, we expect to use the techniques and results in the undegrad education at University of Illinois and Univerity of Texas.

What is the impact on other disciplines?
Nothing to report.

What is the impact on the development of human resources?

The project is providing a rich education environment for future scientists. One of the students involved (Connor Simmons) will be graduating in May 2013 and join the IT technology sector.

What is the impact on physical resources that form infrastructure?
Nothing to report.

What is the impact on institutional resources that form infrastructure?
Nothing to report.

What is the impact on information resources that form infrastructure?
Nothing to report.

What is the impact on technology transfer?

All tools in our toolset are released freely for public use.

What is the impact on society beyond science and technology?

Just as machinery fostered the industrial revolution, machinery automating software evolution tasks will foster a revolution in software technology. By embracing the view that change (transformations) lies at the heart of software development, US software education and industry will take a critical step toward maintaining its supremacy long-term.

Changes

Changes in approach and reason for change
Nothing to report.

Actual or Anticipated problems or delays and actions or plans to resolve them
Nothing to report.

Changes that have a significant impact on expenditures

Nothing to report.

Significant changes in use or care of human subjects
Nothing to report.

Significant changes in use or care of vertebrate animals
Nothing to report.

Significant changes in use or care of biohazards
Nothing to report.