COPE: Vision for a Change-Oriented Programming Environment

Danny Dig
Oregon State University
digd@eecs.oregonstate.edu

Ralph Johnson, Darko Marinov, Brian Bailey
University of Illinois
{rjohnson, marinov, bailey}@illinois.edu

Don Batory
University of Texas
batory@cs.utexas.edu

ABSTRACT

Software engineering involves a lot of change as code artifacts are not only created once but maintained over time. In the last 25 years, major paradigms of program development have arisen – agile development with refactorings, software product lines, moving sequential code to multicore or cloud, etc. Each is centered on particular kinds of change; their conceptual foundations rely on transformations that (semi-) automate these changes.

We are exploring how transformations can be placed at the center of software development in future IDEs, and when such a view can provide benefits over the traditional view. COPE, a *Change-Oriented Programming Environment*, looks at 5 activities: (1) analyze what changes programmers typically make and how they perceive, recall, and communicate changes, (2) automate transformations to make it easier to apply and script changes, (3) develop tools that compose and manipulate transformations to make it easier to reuse them, (4) integrate transformations with version control to provide better ways for archiving and understanding changes, and (5) develop tools that infer higher-level transformations from lower-level changes. Characterizing software development in terms of transformations is an essential step to take software engineering from manual development to (semi-) automated development of software.

1. OUR VIEW OF TODAY

Software constantly changes. Most companies spend more on maintaining old systems than on building new ones. It is often reported that at least two-thirds of software costs are due to evolution [1, 2], with some industrial surveys [3] claiming 90%. This is good, because it is a sign that we build software that is worth keeping. But it means that the traditional view of software development is wrong: software development is not about the conversion of user requirements into *new software* as much as it is about responding to user needs by changing *existing software*. Change is the heart of software development.

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA
 ISBN 978-1-4503-4205-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889208

This observation should sound familiar. In the last 25 years, major paradigms of software development and software platforms have emerged based on (semi-)automatable changes. Agile development is centered on the constant restructuring of programs to meet new demands; behavior-preserving transformations called refactorings lie at its foundation. *Model Driven Engineering (MDE)* focusses on the mapping of high-level models of programs to low-level implementations. Reusable transformations are the computational foundation of MDE. *Software product lines (SPLs)* raise the idea of change to features (stereo-typical increments in program behavior) to explain how simple programs can be modified in structured ways to create program families. Features are program transformations in SPLs. A modern activity is parallelizing sequential programs. This too can be understood as applying a series of behavior-preserving transformations that map sequential computations to parallel computations. In short, disjoint research communities have recognized that stereotypical changes can be (semi-)automated by transformations; this could be a hallmark of future software development paradigms.

Against this backdrop we ask: how can transformations be placed more at the center of software development in today's IDEs? IDEs are often at the front-line of program development today; how can we begin to make changes (transformations) front-and-center in tomorrow's software development and "not get in the way" of modern thinking? Exploring answers to these questions will help to promote the revolution in thinking about transformations in modern software engineering, and to demonstrate concrete benefits of making programmers more productive and software more reliable.

2. OUR VIEW OF TOMORROW

We envision an environment where program transformations unify many of the software development activities. By investigating when, how, and why transformations can be placed at the center of software development, we can lay the foundations of scientific theories and build tools that will improve our abilities to automate and simplify software evolution. We want both average and expert programmers to be able to write, script, modify, and replay their own transformations. We want programmers to think about programs as compositions of transformations, and to automate as many of these transformations as possible.

To this end, we propose to create a *change-oriented programming environment* (COPE). Each change, even manual edits, will be considered a program transformation, since it transforms one version of a program to the next. Thus, in

addition to the traditional view of a program being a collection of modules, COPE will promote an orthogonal view of a program as a sequence of transformations. Programs (and/or their modules) will be objects in COPE, and transformations will be mappings that are applied to these objects. By providing this more abstract view, COPE will make programmers understand and control change better. By mechanizing some changes, transformations will be more reusable, making change less expensive and more reliable.

2.1 Example Killer Applications of COPE

An environment like COPE, which treats changes as first-class entities, enables programmers to perform many development tasks in ways not possible with current technology. Here are three examples.

Ex1. Multi-views. Although some transformations (e.g., refactorings) make code more readable, others (e.g., optimizations or security improvements) make it less readable and more expensive to maintain. Suppose that Alice has to finish two tasks: extend an existing video encoding algorithm to support a new device and fix a security hole. She cannot juggle both tasks at once, and current IDEs show all the code entangled. To disentangle it, Alice relies heavily on COPE's *multiple views*, where each view shows only some code fragments. First she uses a view that shows only the central parts of the encoding algorithm but hides all details of optimizations or security fixes. Next she applies a temporary transformation to produce a new view; e.g., the algorithm is spread over methods in a large class hierarchy, and she defines a view that uses the Visitor design pattern to put all the methods in a single class. Now she can understand the algorithm and change it easier. Then she starts fixing the security hole. She uses a view that shows only the code pertaining to security and hides all other code. She can quickly spot and fix the security hole as the other details are hidden. Different views emphasize the part of the code that Alice cares to see at a given moment, thus *views make code more understandable.*

Ex2. Separate portable from non-portable code. Some transformations (e.g., platform-specific optimizations) make code non-portable. Suppose Bob migrated his desktop email application to support a web interface platform. He had to first undo desktop-specific optimizations and then apply new transformations for the web platform. This was expensive. A recent DOE workshop estimated $100 per line of code ported to another platform. Sadly, in a few years, he will go through the same expensive process to retarget the email application for a smartphone. Instead, Bob uses COPE to support *multiple versions* of his application by controlling the transformations that are applied. For example, the desktop version can use several optimizations but not security transformations. The web version would use security transformations. The smartphone version will use power-aware transformations. Bob always adds new features to one single version of the code, regardless of how many platforms he supports. *This control of transformations will make code more portable.*

Ex3. Manipulate transformations. Carol is a programmer at a US stock exchange that has recently expanded into the Brazilian market. Due to lack of time, the developers forked the code into two branches rather than making a single code base for both markets. However, Carol and her team now frequently face the maintenance problem of pushing some common changes into both branches. Recently, Carol had to change the error-recovery policy which required changing 270 try-catch blocks for the US code branch. She started by making the changes manually, then decided to use COPE to write a script to perform them. The script made it easy to push the changes to the Brazilian branch even though this code uses different names for the code entities. *Scripting and reusing transformations make the programmer more productive.*

Recap. Supporting just one of these tasks would make COPE worthwhile for some programmers. Supporting all such tasks will make COPE worthwhile for all programmers. COPE will make them more in control of their project and more productive. That is why we believe COPE will revolutionize software engineering.

2.2 COPE Challenges

As a foundational paradigm of COPE, we envision programmers thinking about *a software system as a core program followed by a series of transformations.* As a community of software evolution researchers, we need to answer questions about *when, how, and why* this paradigm helps:

When would it be helpful and when would it get in the way of understanding? How often is it really possible to use a transformation to identify the task relevant code? When is it more profitable to encode a change as a reusable transformation, and when to simply edit the code directly?

How would it affect development activities such as testing and debugging? What should be tested: the core program, the transformations? In case of a bug, how to determine which transformation(s) caused the bug? How can COPE co-exist with the traditional view of code/edit-centered IDE? Can it be introduced incrementally? Can it be intermixed with low-level changes?

Why would COPE be more successful than previous approaches, for example, concern-based design or aspects? Are there just a few obvious compelling use cases, or is there a much larger set of use cases that envisions code interactions as centering around transformations? Could (or should) some functionality be modularized as a transformation rather than as parameterized component?

To turn COPE into reality, we address five key challenges:

(1) Understand transformations. We will study how programmers perceive, recall, and communicate change in complex code. Our work [4] addressed some questions for refactorings—e.g., how programmers understand them and why, when, and where programmers do (not) apply them—but this work needs to be significantly extended to handle other types of transformations.

(2) Automate transformations. Writing and scripting transformations today is hard, e.g., implementing a simple transformation such as RenameMethod 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—

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 patch) 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.

To ensure that our results are general enough to handle a wide range of changes, we will study changes in four different exemplars: (1) upgrading applications when third-party components evolve, (2) applying features to programs in a product line, (3) fixing bugs, and (4) evolving tests to match the new expected behavior of a system. Some exemplars focus on changes to behavior; others keep behavior the same. Some stress automation, others manipulation or inference of transformations. Together they define a large design space to explore. We will generalize from these exemplars to build a unified framework, COPE.

3. PRELIMINARY & ONGOING RESULTS

Our previous and ongoing research has explored isolated points in the program transformation space. The success of that research—(1) showing the feasibility and desirability of representing programs as sequences of transformations and (2) developing new techniques and tools integrated in IDEs used by millions of programmers—motivates the proposed deeper exploration of a large space.

3.1 Analytics for fine-grained changes

We need a science of software changes. But this science can only flourish if we develop the theories, models, and instruments to measure and understand changes.

Some researchers may argue that our community has been already analyzing software changes for decades: we have a plethora of data stored in Version Control Systems (VCS) that we can analyze, and there are communities (like the Mining Software Repositories) who are already doing a great job at this. However, we argue that this is the equivalent of reactive (as opposed to proactive) instruments, and the data that we get is too coarse grained and not enough.

We developed CodingTracker [5,6], an Eclipse plugin that records textual edits and operations from the IDE and uses this low-level recording to infer changes on a program represented as Abstract Syntax Trees. We view CodingTracker as the lowest level of COPE. So far we have used CodingTracker to capture more than 1,500 hours of code development from 26 Java programmers. Our experiment highlights severe limitations in using VCS data to analyze code changes.

First, VCS data is incomplete. We found that 35% of the changes that programmers make never get checked into VCS because they are shadowed by other changes (for example when doing several attempts to tune performance). Of these shadowed changes, a few are due to commenting/uncommenting or by undoing changes, but 90% are real code changes.

Second, VCS data is imprecise. A single VCS commit may contain several overlapping changes to the same program entity. For example, a refactored method could also be edited in the same commit.

Third, answering research questions that correlate code changes with other development activities (e.g., test runs, refactoring) is impossible. VCS does not capture many kinds of developer actions, such as running or debugging the application or the tests. This severely limits the ability to study the code development process. How often do developers commit changes that are untested? For developers claiming to use XP and Test Driven Development (TDD), do they write tests before the production code, or after they wrote the production code?

Here are ideas that capture our imagination. First, we want to develop software analytics to better understand code evolution. This can **turn SE into a data-driven field**. For example, at one of the companies that we partnered to do research with, the CTO mandates that all software developers do TDD. Using rich data such as the one from our partner company will help answer question such as: Do certain development practices make the developer more productive? Can we use smart real-time analytics and visualizations to change developer habits? For example, imagine an eCoach that prompts developers with a voice of consciousness: "it seems that you have been developing too much code without running any tests yet".

Second, based on proactive monitoring of changes, we want to develop **recommendations for program changes**. Can we recommend to a developer which APIs to use based on how other developers changed similar code in the past? Can we detect when a developer is in the middle of a complex change, and use the programming environment to interactively complete the rest of the change based on learned changes from the past?

Third, it would **increase the awareness of changes**: imagine that the code is complemented by information and warnings that show the risk of making a change. For example, when a programmer tries to change a certain piece of code, a warning pops up: "these five lines of code that you are planning to change are very complex and the previous developer who worked on them, spent 3 hours fiddling with them", or "you are in the middle of a change, but this change is known to have caused many bugs before".

Our recent work [7] shows that developers use 3 conceptual "lenses" to reason about software changes. Among them, the "immediate lens" describes recent changes that developer made but did not yet commit in the VCS. Inspecting such changes is one of the most acute developer needs but it is not currently met by tools. We envision many applications for this lens: backtracking and selective undo [8] of changes, navigating and grouping changes, and splitting changes into logical units of commits.

3.2 API evolution as program transformations

Ideally, the interface to a software component never changes. In practice, it often does. For example, between two major versions, the Eclipse IDE had 51 API changes that were

not backwards-compatible. Struts, a popular framework for web applications, had 136 API changes [9] that were not backwards-compatible. Such API changes require applications that use components to be changed (upgraded) before new versions can be used. The current state-of-the-practice for component evolution uses text-based tools which are too poor to express the complexities of API evolution. Instead, our solution treated API changes as first-class citizens: we expressed API evolution in terms of program transformations with well-defined semantics. We also defined an algebra [10] to express commutativity and dependences among API changes.

To make our solution practical, we developed tools [11] to detect API program transformations (focusing on API refactorings) and then replay them on the applications (this feature ships with the official release since Eclipse 3.2). Moreover, we developed a versioning tool [12] that semantically merges the library and application API changes. This is the first system that intelligently composes refactorings and edits, two kinds of program transformations.

3.3 Automating program transformations

Refactoring scripts [13, 14]. It is widely known that many design patterns can be created by repeated applications of refactorings. It is long overdue that IDEs support refactoring scripts that programmatically invoke refactorings in repeated and well-understood ways. We have leveraged the Eclipse refactoring engine to build a plug-in that allows programmers to write refactoring scripts [13] in Java; the lack of specific refactorings and refactoring reliability drove us to build a new refactoring engine [14], one that is significantly faster and more extensible than that of Eclipse.

Product lines (current work) We built a set of tools, AHEAD [15], for product line creation that equated features with program transformations. The next generation of these ideas no longer require new programming language constructs [16]; we instead use languages as-is and "color code": all code belonging to the blue feature is painted blue, all code belonging to the red feature is painted red, etc. Code with multiple colors denotes feature interactions. Particular programs of a product line are created by projection (i.e., eliminating code whose features were not selected). Coloring elevates text preprocessing to a disciplined use of optional (AST) program structures, which has a solid algebraic foundation.

Evolving tests [17–19]. Programmers often change software in ways that cause tests to fail. If programmers determine that failures are caused by errors not in the code under test but in the test code itself, they should repair failing tests. Fortunately, simple program transformations can repair many failing tests automatically. Example transformations include replacing literal values in tests, changing assertion methods, or replacing one assertion with several. To automate such transformations, we developed a tool, ReAssert [17]. Experiments show that ReAssert can repair many common test failures and that its suggested repairs correspond to programmers' expectations. Our initial work used several ad-hoc transformation for repair [17], and our later repairs used symbolic execution [18]. We have also developed a technique [19] to transform tests for multithreaded code which showed the importance of using dynamic analysis for test code transformation.

4. CONCLUSIONS

We think it is time for a software revolution in software development, similar with the one in the industrial revolution era, from manual development to semi-automated development of software. By placing **transformations at the center of software development**, we lay the foundations of a change-oriented programming environment (Cope).

By creating a consortium of researchers, we can reuse the platforms and datasets, and avoid spending several years to re-build such platforms. Our call is for collaborators to create a common platform for research for the next 10 years, and produce results that become standard software development practice in 15 years. If Cope is successful, it will have a major impact on industry, as it will influence the major IDEs and software engineering tools to **treat change and transformations as first-class citizens**. Cope will be also a catalyst for education as it can revamp the undergrad software engineering curriculum to emphasize the activities of changing large codebases.

Acknowledgements

We thank the anonymous reviewers for useful feedback. This research is partly funded by the NSF grants CCF-1439957 and CCF-1212683.

5. REFERENCES

[1] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *CACM*, vol. 36, pp. 81–94, 1993.
[2] T. Guimaraes, "Managing application program maintenance expenditures," *CACM*, vol. 26, pp. 739–746, 1983.
[3] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, pp. 17–23, 2000.
[4] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, Disuse, and Misuse of Automated Refactorings," in *ICSE*, 2012.
[5] "COPE Home Page," http://cope.eecs.oregonstate.edu.
[6] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *ECOOP*, 2012.
[7] C. Mihai, S. Srinivasa, D. Dig, and B. Bailey, "Software history under the lens: A study on why and how developers examine it," in *ICSME*, 2015.
[8] Y. Yoon and B. A. Myers, "Supporting selective undo in a code editor," in *ICSE*, 2015.
[9] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," *JSME*, vol. 18, pp. 87–103, 2006.
[10] D. Dig, "Automated upgrading of component-based applications," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2007.
[11] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automatic detection of refactorings in evolving components," in *ECOOP*, 2006.
[12] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen, "Effective software merging in the presence of object-oriented refactorings," *IEEE TSE*, vol. 34, no. 3, pp. 321–335, 2008.
[13] J. Kim, D. Batory, and D. Dig, "Scripting Parametric Refactorings in Java to Retrofit Design Patterns," in *ICSME*, 2015.
[14] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *ICSE*, 2016.
[15] "AHEAD tool suite," www.cs.utexas.edu/users/schwartz/ATS.html.
[16] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE*, 2008.
[17] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting repairs for broken unit tests," in *ASE*, 2009.
[18] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *ISSTA*, 2010.
[19] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, "Improved multithreaded unit testing," in *ESEC/FSE 2011*.