Effective Software Merging in the Presence of Object-Oriented Refactorings

Danny Dig, Kashif Manzoor, Ralph Johnson, *Member*, *IEEE Computer Society*, and Tien N. Nguyen, *Member*, *IEEE*

Abstract—Current text-based Software Configuration Management (SCM) systems have trouble with refactorings. Refactorings result in global changes, which lead to merge conflicts. A refactoring-aware SCM system reduces merge conflicts. This paper describes MolhadoRef, a refactoring-aware SCM system, and the merge algorithm at its core. MolhadoRef records change operations (refactorings and edits) used to produce one version and replays them when merging versions. Since refactorings are change operations with well-defined semantics, MolhadoRef treats them intelligently. A case study and a controlled experiment show that MolhadoRef automatically solves more merge conflicts than CVS while resulting in fewer merge errors.

Index Terms—Refactoring, merging, Software Configuration Management, version control systems.

1 Introduction

One of the important kinds of change in object-oriented programs is refactoring [2]. Refactorings are program transformations that improve the internal design without changing the observable behavior (e.g., renamings, moving methods between classes, changing method signatures). Automated refactoring tools have become popular because they allow programmers to change source code more quickly and safely than manually. However, refactoring tools make particular demands on text-based SCM systems.

SCM systems work best with modular systems. Different programmers tend to work on different modules and, so, it is easy to merge changes. However, refactorings cut across module boundaries and cause changes to many parts of the system. SCM systems signal a conflict when two programmers change the same line of code even if each just changes the name of a different function or variable. So, SCM systems have trouble merging refactorings.

A common process [3] for refactoring on large projects is "check everything in and then wait until refactoring is done," analogous to a "code freeze." However, this serializes the development of a code. In addition, by forcing refactorings to be performed by only a few people at a certain time, opportunities for refactoring are lost.

Although the number of global changes varies from system to system, our previous study [4] of five widely used mature Java components showed a significant number of global changes. For instance, Struts had 136 API changes over a period of 14 months. In each system, more than 80 percent of the API changes were caused by refactorings. Because of lack of support from SCM systems, these changes were tedious to incorporate manually, although a refactoring-aware SCM could have incorporated them automatically.

Text-based SCM systems are unreliable. They report merge conflicts only when two users change the same line of code. However, a merge might result in an incorrect program, even when the changes are not on the same line. This is especially true in object-oriented programs. For instance, if one user renames a virtual method while another user adds a new method in a subclass, even though these changes are not lexically near each other, textual merging could result in accidental method overriding, thus leading to unexpected runtime behavior.

This paper describes MolhadoRef, a refactoring-aware SCM for Java, and the merge algorithm at its core. MolhadoRef has an important advantage over a traditional text-based SCM. MolhadoRef automatically resolves more conflicts (even changes to the same lines of code). Because it takes into account the semantics of refactorings, the merging is also more reliable: There are no compile errors after merging and the semantics of the two versions to be merged are preserved with respect to refactorings.

Correct merging of refactorings and manual edits is not trivial: Edits can refer to old program entities as well as to newly refactored program entities. MolhadoRef uses the *operation-based* approach [5]: It represents a version as a sequence of change operations (refactorings and edits) and replays them when merging. If all edits came before refactorings, it would be easy to merge the two versions by first doing a textual merge and then replaying the refactorings. However, edits and refactorings are mixed, so, in order to commute an edit and a refactoring, MolhadoRef *inverts* refactorings. Moreover, refactorings will sometimes have *dependences* between them.

MolhadoRef uses Eclipse [6] as the front end for changing code and customizes Molhado [7], a framework for SCM, to

- *D. Dig is with MIT CSAIL, The Stata Center, Building 32-G720, 32 Vassar Street, Cambridge, MA 02139. E-mail: dannydig@csail.mit.edu.*
- *K. Manzoor is with Techlogix, Pakistan. E-mail: cashifmanzoor@gmail.com.*
- *R. Johnson is with the Department of Computer Science, Siebel Center, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, IL 61801. E-mail: johnson@cs.uiuc.edu.*
- *T.N. Nguyen is with the Department of Electrical and Computer Engineering, Iowa State University, 3218 Coover Hall, Ames, IA 50011. E-mail: tien@iastate.edu.*

Manuscript received 21 Jan. 2007; revised 5 Oct. 2007; accepted 21 Jan. 2008; published online 22 Apr. 2008.
Recommended for acceptance by B.G. Ryder.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0016-0107.
Digital Object Identifier no. 10.1109/TSE.2008.29.

store Java programs. Although the merging algorithm is independent of the Molhado infrastructure and can be reused with other SCM back ends, building on top of an ID-based SCM like Molhado allows our system to keep track of the refactored entities. When evaluating MolhadoRef on a case study of three weeks of its own development and through one controlled experiment, we found that MolhadoRef merges more safely and more automatically than CVS while never losing the history of refactored entities.

MolhadoRef merges edits using the same three-way merging [8] of text-based SCMs. It is when MolhadoRef merges refactorings that it eliminates merge errors and unnecessary merge conflicts. So, the more that refactorings are used, the more benefits MolhadoRef provides.

This paper makes the following contributions:

- It presents an important problem that appears when merging refactored code in multiuser environments.
- It presents the first algorithm to effectively merge refactorings and edit operations.
- It evaluates the effectiveness of refactoring-aware merging on one real-world case study and one controlled experiment.

Without losing any power to merge manual edits, MolhadoRef converts refactorings from being the weakest link in an SCM system to being the strongest.

2 Motivating Example

To see the limitations of text-based SCM, consider the simulation of a Local Area Network (LAN) shown in Fig. 1. This example is used as a refactoring teaching device [9] in many European universities.

Initially, there are five classes: `Packet`, a superclass `LANNode`, and its subclasses `PrintServer`, `NetworkTester`, and `Workstation`. All `LANNode` objects are linked in a token ring network (via the `nextNode` variable) and they can `send` or `accept` a `Packet` object. `PrintServer` overrides `accept` to achieve specific behavior for printing the `Packet`. A `Packet` object sequentially visits every `LANNode` object in the network until it reaches its addressee.

Two users, Alice and Bob, both start from version V_0 and make changes. Alice is the first to commit her changes, thus creating version V_1, while Bob creates version V_2.

Since method `getPacketInfo` accesses only fields from class `Packet`, Alice moves method `getPacketInfo` from class `PrintServer` to `Packet` (τ_1). Next, she defines a new method, `sendPacket(Packet)` (τ_2), in class `NetworkTester`. The implementation of this method is empty because this method simulates a broken network that loses packets. In the same class, she also defines a test method, `testLosePacket` (τ_3) and implements it to call method `sendPacket` (τ_4). Last, Alice renames `WorkStation.originate(Packet)` to `generatePacket(Packet)` (τ_5). Alice finishes her coding session and commits her changes to the repository.

In parallel with Alice, Bob renames method `PrintServer.getPacketInfo(Packet)` to `getPacketInformation(Packet)` (τ_6). He also renames the polymorphic method `LANNode.send()` to `sendPacket` (τ_7). Last, Bob renames class `WorkStation` to `Workstation` (different capitalization τ_8). Before Bob can commit his changes, he must merge his changes with Alice's.

A text-based SCM system reports merge conflicts when two users change the same line. For instance, because Alice moved the declaration of method (τ_1) while Bob altered the declaration location of the same method through renaming (τ_6), textual merging cannot automatically merge these changes. This is an unnecessary merge conflict because a tool like MolhadoRef which understands the semantics of the changes can merge them.

In addition, because a text-based merging does not know anything about the syntax and semantics of the programming language, even a "successful" merge (e.g., when there are no changes to the same lines of code) can result in a merge error. Sometimes errors can be detected at compile time. For instance, after textual merging, the code in method `testLosePacket` does not compile because it calls method `send`, whose declaration was replaced by `sendPacket` through a rename (τ_7). Such an error is easy to catch, though it is annoying to fix.

Other errors result in programs that compile but have unintended changes to their behavior. For instance, because Alice introduces a new method `sendPacket` in subclass `NetworkTester` and Bob renames the polymorphic method `send` to `sendPacket`, a textual merge results in accidental method overriding. Therefore, the call inside `testSendToSelf` to `sendPacket` uses the empty implementation provided by Alice(τ_2) to simulate loss of packets, while this method call originally used the implementation of `LANNode.send`. Since this type of conflict is not reported during merging or compilation, the user can erroneously assume that the merged program is correct, when in fact the merging introduced an unintended change of behavior.

Fig. 2 shows the merged program after merging with MolhadoRef. MolhadoRef catches the accidental method overriding caused by adding a new method (τ_2) and renaming another method (τ_7) and presents it to the user. The user decides to rename the newly introduced method `NetworkTester.sendPacket` to `losePacket`. This is the only time when MolhadoRef asks for user intervention; it automatically merges all of the remaining changes. The merged version contains all of the edits and refactorings performed by Alice and Bob (e.g., notice that method `getPacketInformation` is both renamed and moved).

3 Background and Terminology

Our approach to refactorings-tolerant SCM systems is based on a different paradigm, called *operation-based merging* [5]. In the operation-based approach, an SCM tool records the operations that were performed to transform one version into another and replays them when updating to that version. An operation-based system treats a version as the sequence of operations used to create it.

Our goal is to provide merging at the API level, that is, our merging algorithm aims for correct usage of all of the APIs. For this reason, we distinguish between operations that affect the APIs and those that do not. MolhadoRef treats a version as being composed of the following three operations: API refactorings, API edits, and code edits. MolhadoRef handles the following API refactorings: rename package, rename class, rename method, move class, move method, and changing the method signature (these were among the most popular refactorings found in previous studies [4]). MolhadoRef handles the following *API edits*: added package, deleted package, added class,

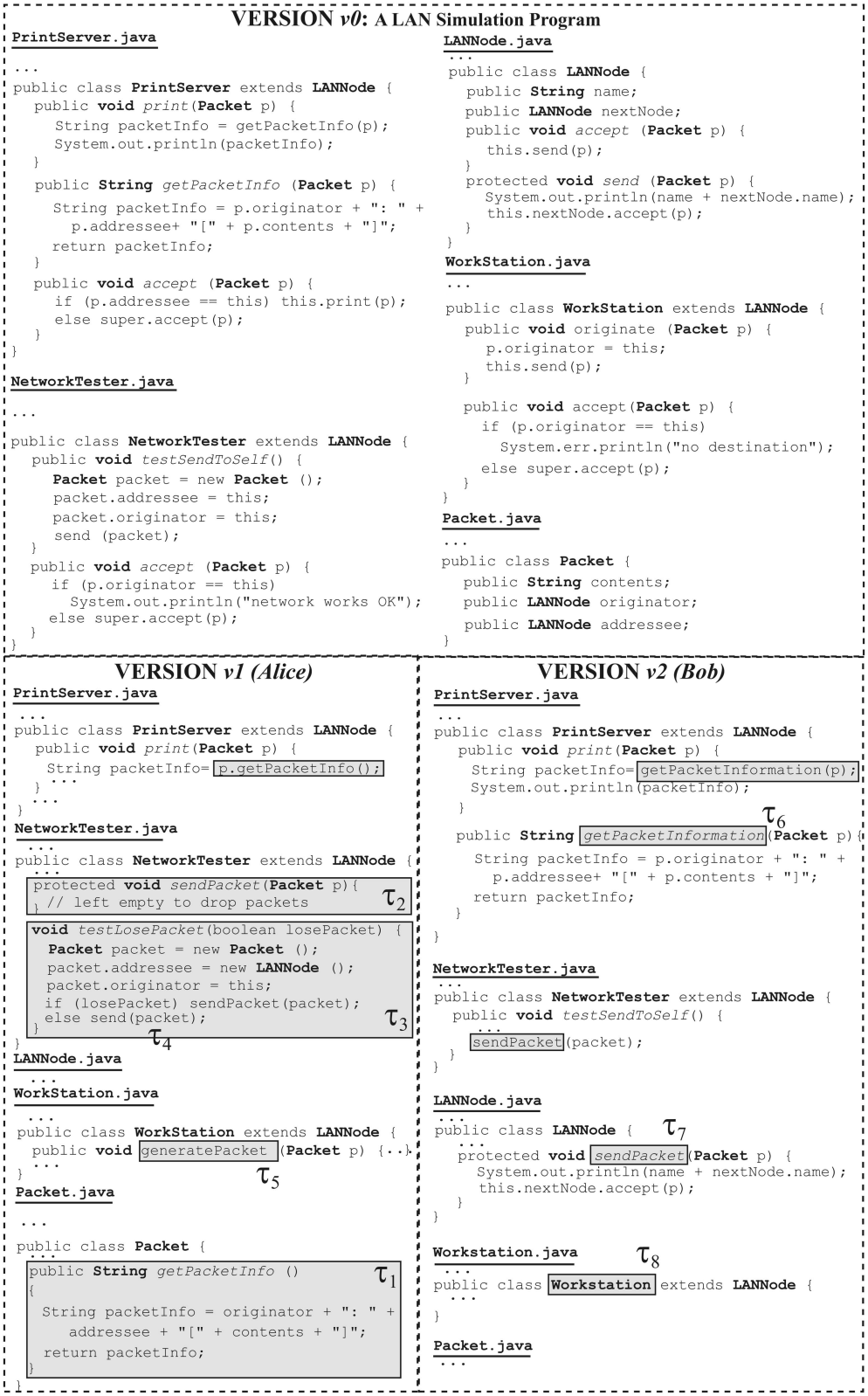

Fig. 1. Motivating example. Versions V_1 and V_2 are created in parallel and are both based on V_0. Shaded boxes show the changes in each version: τ_1 moves method `PrintServer.getPacketInfo` to `Packet`, τ_2 and τ_3 add methods, τ_4 adds a call to `sendPacket`, τ_5 renames `WorkStation.originate` to `generatePacket`, τ_6 renames `PrintServer.getPacketInfo` to `getPacketInformation`, τ_7 renames `LANNode.send` to `sendPacket`, τ_8 renames class `WorkStation` to `Workstation`.

deleted class, added method declaration, deleted method declaration, added field declaration, deleted field declaration. Any other types of edits are categorized as *code edits*.

Code edits do not have well-defined semantics, making it difficult to merge them correctly. API edits have better defined semantics. However, refactorings are the operations

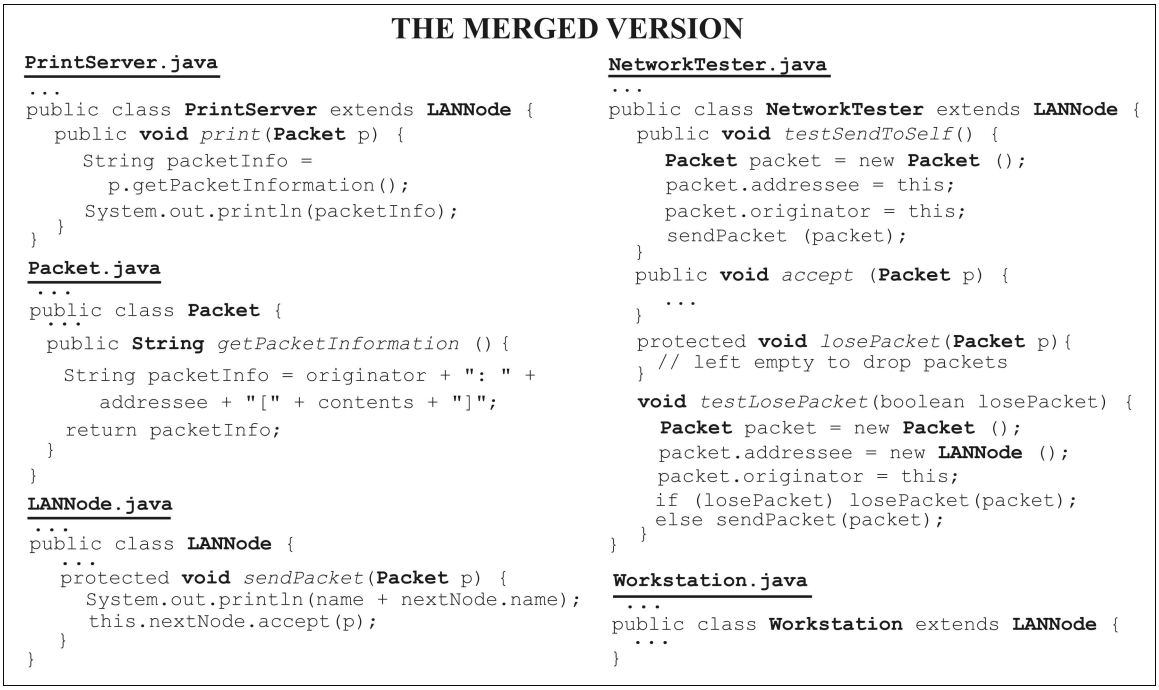

Fig. 2. Resolved motivating example using MolhadoRef.

with the most well-defined semantics and thus are the ones that can benefit the most from operation-based merging. Therefore, MolhadoRef merges code edits textually and, since it is aware of the semantics of refactorings and API edits, it merges them semantically.

Any operation can be regarded as a function from programs to programs, more precisely, a source-to-source program transformation: $\tau : Program \rightarrow Program$

When necessary, we make the distinction between refactorings (ρ) and edits (σ). Refactorings are transformations that preserve the semantics, while edits usually change the semantics of programs.

Operations usually have preconditions. Adding a method to a class requires that the class exists and does not already define another method with the same name and signature, while changing the name of a method requires that the new name is not in use. Applying an operation τ inappropriately to a program P results in an invalid program, represented by $\perp$. The result of applying an operation to $\perp$ is $\perp$:

$$\tau(P) = \begin{cases} P' & \text{if preconditions of } \tau \text{ hold} \\ \perp & \text{if preconditions of } \tau \text{ do not hold.} \end{cases} \quad (1)$$

The application of two operations is modeled by the function composition, denoted by ";". ";" also models the precedence: $\tau_i;\tau_j$ means first apply τ_i and then apply τ_j on the result: $\tau_i;\tau_j(P) = \tau_j(\tau_i(P))$.

Definition 1. *Two operations* commute *on a program P if applying them in either order produces the same valid program P'': $\tau_j;\tau_i(P) = \tau_i;\tau_j(P) = P'' \wedge P'' \neq \perp$.*

Definition 2. *Two operations* conflict *with each other if applying them in either order produces an invalid program: $\tau_j;\tau_i(P) = \perp \wedge \tau_i;\tau_j(P) = \perp$.*

For example, adding two methods with the same name and signature in the same class results in a conflict.

Definition 2 describes conflicts that produce compile errors. MolhadoRef also catches some conflicts that produce runtime errors. MolhadoRef currently catches conflicts that involve method overriding, such as the accidental method overriding between τ_2 and τ_7.

When two operations do not commute for a program P, we say that there is an *ordering dependence* between them. We denote this ordering dependence with the $\prec_P$ symbol.

Definition 3. *τ_j* depends *on $\tau_i (\tau_i \prec_P \tau_j)$ if τ_j and τ_i do not commute:*

$$\tau_i \prec \tau_j \text{ iff } \tau_i;\tau_j(P) \neq \perp \wedge \big(\tau_i;\tau_j(P) \neq \tau_j;\tau_i(P)\big).$$

The $\prec_P$ dependence is *strict* partial order, that is, it is irreflexive, asymmetric, and transitive.

An example of dependence is the renaming of method `WorkStation.originate` to `generatePacket` done by Alice (τ_5) and the renaming of class `WorkStation` to `Workstation` done by Bob (τ_8). If τ_8 is played first, the replaying of τ_5 is not possible because, at this time, the fully qualified name `WorkStation.originate` no longer exists, thus $\tau_5 \prec_P \tau_8$.

This dependence between τ_5 and τ_8 exists because current refactoring engines are based on the names of the program entities and class `WorkStation` no longer exists after replaying τ_8. If the refactoring engine used the IDs of the program elements, changing the names would never pose a problem [10]. To make name-based refactoring engines be ID-based requires rewriting the engine. This is unfeasible, so the next best solution is to *emulate* ID-based engines.

To make the current name-based refactoring engines emulate ID-based ones, there are at least two approaches. The first is to reorder the refactorings (e.g., rename method `WorkStation.originate()` before renaming class

```
1
2  INPUT = {V_2, V_1, V_0, refactoringLogs}
3
4  //Step #1 Detecting Operations
5  Operations ops= 3-wayComparison(V_2,V_1,V_0)
6  Operations refactorings= detectRefactorings(refactoringLogs)
7  Operations edits= detectEdits(ops, refactorings)
8
9  //Step #2 Detecting and Solving Conflicts and Circular Dependences
10 repeat{
11   Conflicts conflicts = detectConflicts(edits, refactorings)
12   {edits, refactorings}= userSolvesConflicts({edits, refactorings}, conflicts)
13   Graph operationsGraph = createDependenceGraph(refactorings, edits)
14   {refactorings, edits, operationsGraph} = userRemovesCircularDependences(operationsGraph)
15 } until noConflictsOrCircularDependences(refactorings, edits, operationsGraph)
16
17 //Step #3 Inverting Refactorings
18 Version V_1_minusRefactorings= invertRefactorings(V_1, refactorings)
19 Version V_2_minusRefactorings= invertRefactorings(V_2, refactorings)
20
21 //Step #4 Textual Merging
22 Version V_merged_minusRefactorings= 3-wayTextualMerge(V_2_minusRefactorings,V_1_minusRefactorings,V_0)
23
24 //Step #5 Replaying Refactorings
25 Operations orderedRefactorings= refactoringsTopologicalSort(operationsGraph)
26 Version V_merged= replayRefactorings(V_merged_minusRefactorings, orderedRefactorings)
27
28 OUTPUT = {V_merged}
```

Fig. 3. Overview of the merging algorithm.

`WorkStation`). The second is to modify the refactoring engine so that, when it changes source code, it also changes subsequent refactorings. For example, during the replay of renaming class `WorkStation` to `Workstation`, the refactoring engine changes the subsequent refactoring `RenameMethod(WorkStation.originate, generatePacket)` to `RenameMethod(Workstation.originate, generatePacket)`. Our merging algorithm uses both approaches.

Consider a scenario where Alice renames method `m1` in superclass `A` (call it operation ρ_1) and Bob desires to override `A.m1` by adding a method `m1` in subclass `B` (call it operation σ_1). Applying these two operations in either order produces a valid program. However, only one order preserves Bob's intent: Applying the edit followed by renaming the method in the superclass preserves the overriding relationship since the renaming ρ_1 also updates the edit σ_1 (renaming a method updates all overriding methods in a class hierarchy). The other order, the renaming ρ_1 followed by the edit σ_1 would result in a program that compiles, but `B.m1` no longer overrides the superclass method, violating Bob's intent. Thus, there is a dependence $\sigma_1 \prec_P \rho_1$.

Definitions 1-3 are mutually exclusive and cover all the cases.

4 Merging Algorithm

4.1 High Level Overview

We illustrate the merging algorithm (see the pseudocode in Fig. 3) using the LAN simulation example presented earlier. The details of each module are found in the later sections.

The merging algorithm takes as input three versions of the software: Version V_0 is the base version and V_1 and V_2 are derived from V_0. In addition, the algorithm takes as input the refactorings that were performed in V_1 and in V_2. These refactoring logs are recorded by Eclipse's refactoring engine. The output is the merged version, V_{merged}, that contains edits and refactorings performed in V_1 and V_2.

Step #1 detects the changes that happened in V_1 and V_2 by performing a three-way comparison between V_1, V_2, and V_0. From these changes and the refactoring logs, it categorizes edits and refactorings operations. For example, in V_1, it detects two added methods, τ_2 and τ_3. In V_2, it detects no edits but only refactorings.

Step #2 searches for compile and runtime conflicts in API edits and refactorings. In our example, it detects a conflict between the add of a new method, τ_2 in V_1, and the rename method refactoring, τ_7 in V_2. This conflict reflects an accidental method overriding. The conflict is presented to the user, who resolves it by choosing a different name for the added method (in this case `losePacket` instead of `sendPacket`). The algorithm also searches for possible circular dependences between operations performed in V_1 and operations in V_2. If any are found, the user deletes one of the operations involved in the cycle (in our example, there are no circular dependences). This process of detecting/solving continues until no more conflicts or circular dependences remain.

Step #3 inverts each refactoring in V_1 and V_2 by applying another refactoring. For instance, it inverts the move method refactoring τ_1 by moving method `getPacketInfo` back to `PrintServer` and it inverts the rename class refactoring τ_8 by renaming `Workstation` back to `theorem`. By inverting refactorings, all of the edits that were referencing the refactored program entities are still kept in place but changed to refer to the old version of these entities. This step produces two software components, $V_1^{-Refactorings}$ and $V_2^{-Refactorings}$, which contain all of the changes in V_1, respectively V_2, except refactorings.

Step #4 merges textually (using the classic three-way merging [8]) all of the API and code edits from $V_1^{-Refactorings}$ and $V_2^{-Refactorings}$. Since the refactorings were previously inverted, all same-line conflicts that would have been

caused by refactorings are eliminated. For example, inside `PrintServer.print` there are no more same-line conflicts. Therefore, textual merging of code edits can proceed smoothly. This step produces a software component, called $V_{\text{merged}}^{-\text{Refactorings}}$.

Step #5 replays on $V_{merged}^{-Refactorings}$ the refactorings that happened in V_1 and V_2. Before replaying, the algorithm reorders all of the refactorings using the dependence relations. Replaying the refactorings incorporates their changes into $V_{merged}^{-Refactorings}$, which already contains all of the edits. For example, replaying a method renaming refactoring updates all of the call sites to that method, including the ones introduced by edits.

4.2 Detecting Operations

To detect refactorings, API edits, and code edits, the algorithm takes as input the source code of the three versions V_0, V_1, and V_2, along with their refactoring logs. Recent extensions to refactoring engines (e.g., [11]) log the refactorings at the time they were performed. This log of refactorings is saved in a configuration file and is stored along with the source code. Since our algorithm is implemented as an Eclipse plug-in, it has access to this log of refactorings. Even in cases when such a log of refactorings is not recorded, it can be detected using RefactoringCrawler [12], a tool for inferring refactorings.

To detect the API edits and code edits, the algorithm employs a three-way textual *comparison* (a two-way comparison cannot distinguish between additions and deletions [13]). This comparison (line 5 in Fig. 3) detects lines, files, and folders that were changed. From this low level information, the algorithm constructs (line 7) the higher level semantic API edits (e.g., add method) by parsing and correlating the positions of changed tokens with that of changed lines.

Even though the scope of our merging is at the API level, to correctly signal compile or runtime conflicts, the algorithm detects a few edit operations that are below the API level. These include add/delete method call, add/delete class instantiation, add/delete class inheritance, and add/delete typecast. From this information, the algorithm is able to detect conflicts like the one appearing if Alice deletes the method declaration `accept` and Bob adds a method call to `accept`.

Some of the edit operations overlap with or are the side effects of refactorings. For example, after renaming class `WorkStation` to `Workstation`, a textual comparison renders `WorkStation` as deleted and `Workstation` as added. Using the information from the refactoring logs, the algorithm discards these two edits since they are superseded by the higher level refactoring operation.

The output of this step is the list of change operations (refactorings and edits) that happened in each of V_1 and V_2.

4.3 Detecting and Solving Conflicts and Circular Dependences

Detecting conflicts. Next, MolhadoRef detects conflicts between operations in V_1 and V_2. For this, it uses a *conflict matrix* (the companion technical report [14] describes all of its cells). For any two kinds of operations, the matrix gives a predicate that responds whether the operations conflict. This matrix includes refactorings, API edits, and the code edits that are currently handled. MolhadoRef instantiates the conflict matrix for the concrete operations detected in the previous step and signals any conflicts between these operations.

$$
\begin{array}{l}
hasConflicts(RenameMethod(m_1, m_2), RenameMethod(m_3, m_4)) := \\
((m_1 = m_3 \vee overrides(m_1, m_3)) \wedge (simpleName(m_2) \neq simpleName(m_4))) \\
\vee \\
((m_1 \neq m_3 \wedge \neg overrides(m_1, m_3)) \wedge (m_2 = m_4 \vee overrides(m_2, m_4)))
\end{array}
$$

Fig. 4. The RenameMethod/RenameMethod cell in the conflict matrix.

Next, we present the content of one single cell in the conflict matrix, namely the case when τ_i is $\text{RenameMethod}(m_1, m_2)$ and τ_j is $\text{RenameMethod}(m_3, m_4)$. These two renamings result in a conflict if 1) the source of both refactorings is the same program element (e.g., $m_1 = m_3$) but their new names would be different or 2) the sources of both refactorings are different program elements but the destination of both refactorings is the same program element (e.g., $m_2 = m_4$). In addition, due to polymorphic overriding, we must also consider the case when two methods are not the same program element, but one method overrides the other.

When the sources of both refactorings are the same (item 1), if methods m_1 and m_3 are in the same class, there would be a compile-time conflict since the users want to rename the same method differently. If the methods m_1 and m_3 are overriding each other, renaming them differently results in a runtime conflict because the initial overriding relationship would be broken. When the destination of the two refactorings is the same (item 2), if methods m_1 and m_3 are in the same class, renaming them to the same name results in a compile-time error (two methods having the same signature and name). If methods m_1 and m_3 are not in the same class and do not initially override each other, renaming them to the same name results in a runtime conflict because of accidental method overriding.

More formally, using first-order predicate logic, Fig. 4 describes the content of the RenameMethod/RenameMethod cell in the conflict matrix. Similar formulas describing the remaining cells in the matrix are in a companion technical report [14]. The predicates in each cell are computed "by hand," but they are carefully revised.

Although the matrix and its predicates describe operations independent of a particular program, in this step, MolhadoRef instantiates it for the program under analysis and its concrete operations detected in Step #1. For example, MolhadoRef detects the accidental method overriding conflict between τ_2 and τ_7 in the motivating example. This conflict is presented to the user who can decide how to solve it.

Detecting circular dependences. In this step, the merge algorithm also creates the dependence graph (line 13 in Fig. 3) between operations performed in the two versions to be merged. Initially, there is a total (i.e., linear) order of the change operations in each version, given by the time sequence in which these operations were applied. However, when merging, the operations can be replayed in any order unless there is a dependence between them. Thus, the total order can be ignored in favor of a partial order, induced by the $\prec_P$ relation.

To create this partial order, we represent each operation as a node in a directed graph, called the *dependence graph*. When $\tau_i \prec_P \tau_j$, the algorithm adds a directed edge from τ_i

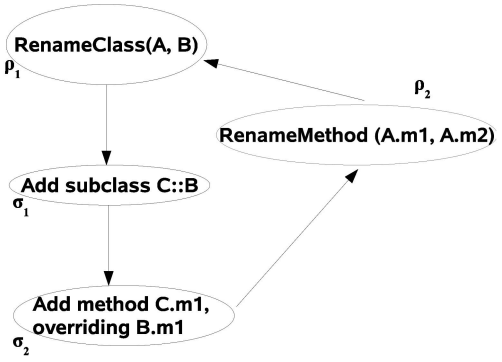

Fig. 5. Circular dependence between operations from two users. Arrows represent the direction of the dependences (e.g., add method C.m1 must be applied before renaming method A.m1). The left-hand side shows operations from Bob, right-hand side shows operations from Alice. The figure depicts the dependence graph as it is created in Step #2 of the algorithm.

to τ_j. To find out the $\prec_P$ dependences, the algorithm uses a *dependence matrix*, which describes dependence relations between all kinds of operations (similar to how the *conflict matrix* describes conflicts). MolhadoRef instantiates the dependence matrix for the concrete operations in the versions to be merged. MolhadoRef places all of the concrete operations in the dependence graph and adds dependence edges using the information from the dependence matrix.

Next, the algorithm searches for cyclic dependences in the dependence graph. There can only be cycles between operations from different users, not between operations from the same user, because, for each user, it was initially possible to play all of the operations. Fig. 5 shows a scenario where a cycle appears between operations from two users. Initially, the base version contains one class `A` with one method `A.m1`. The operations on the left-hand side are performed by Bob and the ones on the right-hand side are performed by Alice. Bob renames class `A` to `B`, then adds a subclass `C` of `B`. Next, in class `C`, Bob adds a method with name `m1` which overrides the method in the superclass `B`. In parallel with Bob, Alice renames method `A.m1` to `m2`.

The arrows on Bob's side indicate the original order in which the operations took place. The arrow from Alice's rename method to Bob's renaming the class points to a dependence caused by the current refactoring engines. The refactoring engines use the fully qualified names to identify the program elements to be refactored; therefore, renaming the method `A.m1` must be performed before renaming its class declaration `A`; otherwise, the refactoring engine can no longer find the element `A.m1`. The arrow from Bob's adding method `C.m1` to Alice's renaming the superclass method A.m1 points to another dependence. The subclass method must be added before the renaming of the superclass method `A.m1` such that, when replaying the `RenameMethod(A.m1, m2)`, it also renames `C.m1` (playing them in a different order would cause the two methods to no longer override each other).

After it finds all cycles, MolhadoRef presents them to the user, who must choose how to eliminate cycles (see the next section). Assuming that there are no more cycles, all operations are in a directed acyclic graph.

User-assisted conflict and dependence resolution. Circular dependences and compile and runtime conflicts require user intervention. To break circular dependences, the user must select operations to be discarded and removed from the sequence of operations that are replayed during merging. Discarding refactorings has no effect on the semantics of the merged program because refactorings are transformations that do not change the semantics. Discarding edits can potentially affect the semantics of the merged program. However, this solution would be used only in extreme cases (we have never run into such a scenario during evaluation). Alternatively, most circular dependences (including the one in Fig. 5) can be solved automatically by MolhadoRef by inverting the refactorings (see Section 4.4).

To solve the syntactic or semantic conflicts caused by name collision, the user must select a different name for one of the program elements involved in the conflict. In our LAN motivating example (Fig. 1), Alice renames method `send` to `sendPacket` and Bob adds a new method declaration `sendPacket` such that the two methods accidentally override each other. This conflict is brought to Bob's attention, who can either choose a different name for his newly introduced method or can pick a new name to supersede the name chosen by Alice. In the motivation example, Bob chose to rename his newly introduced method `sendPacket` to `losePacket`. Once Bob chooses the new name, MolhadoRef automatically performs this rename.

The process of finding and solving conflicts and circular dependences is repeated until there are no more conflicts or circular dependences (line 15 in Fig. 3). The algorithm always converges to a fixed point because it starts with a finite number of operations and the user deletes some in each iteration.

4.4 Inverting Refactorings

Step #3 makes a version of V_1 and V_2 without any refactorings by inverting all refactorings. Inverting a refactoring ρ_1 involves creating and applying an *inverse* refactoring. ρ_1^- is an inverse of ρ_1 if $\rho_1^-(\rho_1(P)) = P$ for all programs P that meet the precondition of ρ_1. For example, the inverse of the refactoring that renames class `A` to `B` is another refactoring that renames `B` to `A`, the inverse of a move method is another move method that restores the original method, the inverse of the extract method is the inline method, the inverse of pull-up member in a class hierarchy is push-down member. In fact, many of the refactorings described in Fowler et al.'s refactoring catalog [2] come in pairs: a refactoring along with its opposite (inverse) refactoring.

Given any refactoring, there exists another refactoring that inverts (undoes) the first refactoring (although such an inverse refactoring cannot be always applied because of later edits). There is an important distinction between what we mean by inverting a refactoring and how the popular refactoring engines (like Smalltalk RefactoringBrowser, Eclipse, or IntelliJ IDEA) undo a refactoring. To decrease memory usage and avoid recomputations of preconditions, the refactoring engines save the location of all source code that was changed by the refactoring. When undoing a refactoring, the engines undo the source changes of these locations.

Although efficient, this approach has a drawback: The only way to undo a refactoring that was followed by edits is to first undo all of the edits that come after it. This approach

is not suitable for MolhadoRef. MolhadoRef must be able to undo a refactoring without undoing later operations. Thus, MolhadoRef inverts refactorings by creating and executing an inverse refactoring operation (which is another refactoring).

To create the inverse refactoring, MolhadoRef uses the information provided in the refactoring logs of Eclipse. Each refactoring recorded by Eclipse is represented by a refactoring descriptor which contains enough textual information to be able to recreate and replay a refactoring. Among others, the descriptor contains information such as what kind of refactoring is created, what the program element on which it operates is, and what other parameters have been provided by the user through the UI (e.g., the new name in case of a rename refactoring or the default value of an argument in case of refactoring that changes a method signature by adding an argument). Out of this information, MolhadoRef creates another refactoring descriptor that represents the inverse refactoring. For example, the inverse of the move method refactoring descriptor representing τ_1 in our LAN example is another move method descriptor representing a refactoring that moves method `Packet.getPacketInfo` back to `PrintServer`.

From the inverse refactoring descriptor, MolhadoRef creates and initializes an Eclipse refactoring object. Once the refactoring object is properly initialized, the refactoring is executed using Eclipse's refactoring engine.

Inverting a refactoring and executing the inverse refactoring also changes the edits. Recall the motivation example where Bob renames method `getPacketInfo` to `getPacketInformation`. Later, he adds a new method call to `getPacketInformation`. By inverting the rename method refactoring with the inverse refactoring (renaming `getPacketInformation` to `getPacketInfo`), the new call site to `getPacketInformation` is also updated while keeping the call site in the same place. Deleting the call site altogether would have introduced a different behavior, while leaving the call site untouched would have produced a compilation error.

Probably, the most notable aspect of inverting refactorings is that it inverts the dependences between edits and refactorings, allowing refactorings to come after edits; thus, it changes a $refactoring \prec_P edit$ dependence into $edit \prec_P refactoring$ dependence. This has two advantages. First, when replaying refactorings in Step #5, the fact that refactorings come after edits ensures that all changes caused by refactorings are incorporated into edits. Second, inverting refactorings automatically breaks most of the circular dependences between operations. Recalling the example from Fig. 5 with circular dependence, Fig. 6 shows the dependence graph after inverting the rename class refactoring. Notice that the edits now refer to class `A` instead of `B` and there are no more circular dependences.

Just as refactorings have preconditions, inverting a refactoring also has preconditions and, if those preconditions are not met, then a refactoring cannot be inverted. Edits in V_1 and V_2 that were applied after refactorings could break the preconditions of inverse refactorings. To handle such cases, we have three heuristics: adding program transformations, storing additional information before inverting a refactoring, or a fallback heuristic in case the others fail.

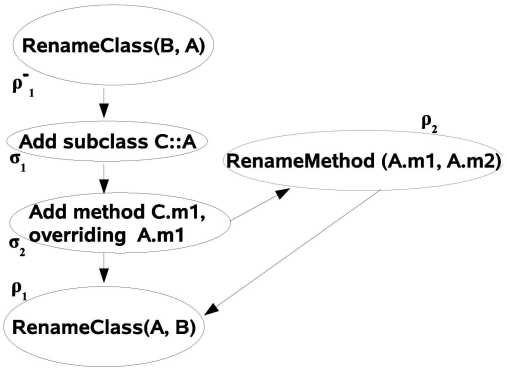

Fig. 6. Resolved example of circular dependence from Fig. 5. This figure shows the dependence graph *after* Step #3 of the algorithm. Applying the inverse refactoring (ρ_1^-) effectively pushes ρ_1 after the edits, thus breaking the circular dependence.

Heuristic #1: Renaming a program element. This heuristic renames a program element to a unique name when name collisions prevent inverting a refactoring. For example, if Bob renames `PrintServer.getPacketInfo` to `getPacketInformation` (τ_6) and later adds a new method `getPacketInfo` in the same class, inverting the rename refactoring in Step #3 is not possible because the name `getPacketInfo` is already in use (by the lately added method). MolhadoRef searches for potential name collisions before inverting the refactoring and executes another renaming to avoid the collision. In this case, before inverting the refactoring, the algorithm renames the newly introduced method `getPacketInfo` to a unique name, say `getPacketInfoABC123`, and tags this rename refactoring. Now that there are no more name collisions, renaming τ_6 can be inverted. Later, in Step #5, after all of the regular refactorings have been replayed, the algorithm inverts all refactorings marked with tags. Thus, it renames the new method `getPacketInfoABC123` back to `getPacketInfo`. At this stage, there are no more name collisions because τ_6 would have executed.

Heuristic #2: Storing additional information. This heuristic stores additional information before inverting a refactoring since some information can get lost when inverting refactorings. Consider the case when Bob changes the signature of a method `sendPacket` by adding an extra argument of type integer with the default value 0 to be used in method calls. Later, he adds a call site where he passes the value 7 for the extra argument. Inverting the refactoring and replaying it naively would lose the value 7 and replace it with value 0. Before inverting the refactoring, MolhadoRef saves the location of the new call sites and the values of parameters so that it can restore the same values later when replaying the refactoring in Step #5.

Heuristic #3: Fallback—treating refactorings as edits. When no heuristic for inverting a refactoring is found, the algorithm treats the refactoring as a classic textual edit, namely, the refactoring is not inverted and replayed, but its code changes are incorporated by textual merging. Although the advantages of incorporating the semantics of the refactoring are lost, the algorithm can always make progress and, in the worst case, it is as good as classic textual merging.

The first two heuristics are sufficient to invert all of the refactorings in the evaluations we have done. Nevertheless,

further evaluations might require developing new heuristics to handle other types of refactorings or force us to use the fallback heuristic (heuristic #3). An analysis of all refactorings currently supported in Eclipse shows that all of these refactorings could be inverted by first renaming conflicting program elements (heuristic #1) or storing additional information before inverting the refactoring (heuristic #2).

4.5 Textual Merging

Once refactorings are inverted, all of the edits in V_1 and V_2 that referred to the refactored APIs now refer to the APIs present in version V_0. The algorithm merges textually all files that were changed by edits using the three-way merging [8] that most text-based SCMs use.

All code changes inserted by refactorings that would have caused same-line or same-block conflicts are eliminated due to the fact that refactorings were previously inverted. In our LAN example, although both users changed the declaration of `getPacketInfo` (τ_1 and τ_6), after inverting the refactorings, the call to method `getPacketInfo` inside `PrintServer.print` no longer causes same-line conflict.

Still, if two users change the same lines by code edits (not refactorings), this can generate a same-line conflict requiring user intervention, although MolhadoRef can automatically merge a few more edits than textual-based merging. For example, if Alice and Bob each add a new method declaration at the same position in a source file, MolhadoRef merges this automatically using the semantics of API edits. In contrast, textual-based merging would signal a same-line conflict. However, it is when multiple refactorings affect the same lines that MolhadoRef shines over text-based merging.

4.6 Replaying Refactorings

Current refactoring engines identify program entities with fully qualified names. Within a stream of operations from a single version, names will always be consistent because each refactoring works with the current names of program elements. However, when refactorings are merged from two different streams, renamings can interfere with each other in two ways.

The first is where the refactorings refer to two different entities, but one of them has a name that includes the other. For example, the fully qualified name of a method includes the name of its class. If one refactoring renames a class and the other changes a method in that class, it is important that the right method gets changed. MolhadoRef solves this problem by making sure that the refactorings of a method are performed before the refactorings that rename its class. More precisely, MolhadoRef uses a topological sort algorithm [15] to reorder the nodes in the refactorings DAG created in Step #2.

The second is where two refactorings refer to the same entity. Sometimes, this is a conflict that must be resolved by the user, such as when the two refactorings change the name of the same entity. This case would have been resolved by Step #2. So, the only remaining cases are when the two refactorings change the same entity, but in different ways. For example, one refactoring could rename a method and the other could move it to a new class (e.g., τ_6 and τ_1). Changing either the method name or the class name will invalidate the other refactoring. MolhadoRef solves this problem by modifying refactorings. If a refactoring from one version is replayed after a rename or a "move method" refactoring from the other version, second refactoring is changed to use the new name. This lets a name-based system like Eclipse emulate an ID-based system.

To handle multiple refactorings to the same element, we extended the definition and semantics of a refactoring. In addition to source code, a refactoring changes subsequent refactorings in a chain of refactorings. An *enhanced* refactoring is a transformation from source code and a chain of refactorings to another source code and another chain of refactorings. Conceptually, our enhanced refactoring, $\rho^{Enhanced}$, is the pair of a classic refactoring transformation, ρ, with another transformation, θ, that changes subsequent refactorings in a chain:

$$\theta : Refactorings \rightarrow Refactorings$$
$$\rho^{Enhanced} =< \rho, \theta > .$$

Composing an enhanced refactoring with another refactoring changes the second refactoring:

$$\rho_i^{Enhanced}; \rho_j =< \rho_i, \theta_i > \rho_j = \rho_i; (\theta_i(\rho_j)).$$

Each θ transformation is dependent upon the type of enhanced refactoring from which it is a part. For instance, a θ_{Ren} transformation applied on a move refactoring changes the parameters of the move refactoring:

$$\theta_{Ren(m \rightarrow k)}(\rho_i) = \begin{cases} Mov(k \rightarrow p) & \text{if } \rho_i = Mov(m \rightarrow p) \\ Mov(z \rightarrow p) & \text{if } \rho_i = Mov(z \rightarrow p). \end{cases} \quad (2)$$

Applying θ_{Ren} on an empty chain of refactorings is equivalent to applying an identity function:

$$\rho_{Ren}^{Enhanced}([\,]) = \rho_{Ren}; (\theta_{Ren}([\,])) = \rho_{Ren}.$$

Given a chain $C = [\rho_i, \rho_{i+1}, \ldots, \rho_k]$, applying a θ transformation on the whole chain C incorporates the effect of the renaming into the whole chain:

$$\theta_{Ren}([\rho_i, \rho_{i+1}, \ldots, \rho_k]) =$$
$$(\theta_{Ren}(\rho_i)); (\theta_{Ren}(\rho_{i+1})), \ldots, (\theta_{Ren}(\rho_k)).$$

The presence of θ transformations elegantly solves cases when multiple refactorings affect the same program element. Revisiting our motivating example, consider the composition of two enhanced refactorings, a rename (τ_6) and a move method (τ_1), that change the same program element, `PrintServer.getPacketInfo`. Each enhanced refactoring is decomposed into the classic refactoring and its θ transformation. Suppose that the rename method is applied first. The enhanced rename method refactoring changes the arguments of the subsequent move method so that the move method refactoring operates upon the new name of the method, `PrintServer.getPacketInformation`.

5 Controlled Experiment

We want to evaluate the effectiveness of MolhadoRef in merging compared to the well-known text-based CVS. For this, we need to analyze source code developed in parallel that contains both edits and refactorings. Software developers know about the gap between existing SCM repositories and refactorings tools. Since developers know what to

TABLE 1
Demographics of the Participants

	Mean	Std.Dev.	Min.	Max
Years Programming	8.35	1.97	5	12
Years Java Programming	4.7	1.72	2.5	7.5
Years Using Eclipse	2.1	1.24	0.5	4

avoid, notes asking others to check in before refactorings are performed are quite common. Therefore, it is unlikely that we will find such data in source code repositories. As a consequence, we designed a controlled experiment.

5.1 Hypotheses

Operation-based merging has intuitive advantages over text-based merging since it is aware of the semantics of change operations. We hypothesize that operation-based merging is more effective than text-based merging. Namely, MolhadoRef:

- **H1** automatically solves more merge conflicts,
- **H2** produces code with fewer compile-time errors,
- **H3** produces code with fewer runtime errors,
- **H4** requires less time to merge

than CVS.

5.2 Experiment's Design

We wanted to recreate an environment, similar to the regular program maintenance, where developers add new features, run regression test suites, and make changes in the code base in parallel. However, we wanted an environment where software developers did not know and worry about other people working on the same code base. We randomly split 10 software developers into two groups, G1 and G2, each group containing five developers. Each developer in group G1 was asked to implement feature ACK, while each developer in group G2 implemented feature MAX_HOPS. All developers were given the same starting code base. At the end, we took their code, stored it in both CVS and MolhadoRef, and merged their changes using Eclipse's CVS client and MolhadoRef.

5.2.1 Demographics

We asked 10 graduate students at UIUC to volunteer in a software engineering controlled experiment. We were specifically looking for students who 1) had extensive programming experience, 2) had extensive Java programming experience, and 3) were familiar with the Eclipse development environment. Table 1 shows the distribution of our subject population. Most subjects had some previous industry experience, two of them were active Eclipse committers, another one was a long-time industry consultant.

We asked the subjects not to work for more than 1 hour. For participating in the study, the subjects were rewarded with a free lunch. During the study, the subjects did not know who the other participants were or what we were going to do with their code. We told them to implement the task as if this was their regular development job. When we got back their solutions, their implementations were not at all similar.

5.2.2 Tasks

Each subject received the initial implementation of the LAN simulation used in our motivating example, along with a passing JUnit test case that demonstrated how to instantiate the system. The system was packaged as an Eclipse project; therefore, the subjects had to use the Eclipse development environment. Along with the system, the subjects received a one-page document describing the current state of the system and the new feature they had to implement. We asked them to write another JUnit test case that exercises the feature they just implemented. We also gave them the freedom to change the current design if they did not like it by using the automated refactorings supported in Eclipse. However, only the feature and adding a test case were mandatory; refactoring was optional.

Task ACK required the subject to change the LAN simulation so that when a destination node received a packet, it sends an `Acknowledgement` packet back to the sender. The `Acknowledgement` packet should have its contents set to "ACK."

Task MAX_HOPS required the subject to fix a problem. In the current implementation, the `Packet` may keep on traveling forever if the destination node does not exist. To solve the problem, if a `Packet` has been traveling around long enough without being consumed by any Node, then it gets dumped/eaten up. "Long enough" represents the maximum number of nodes that a `Packet` is allowed to visit and this needs to be specified by the user.

5.2.3 Variables

Controlled variables. All subjects used Eclipse and Java. Subjects started from the same version and had to implement one of the two tasks. All mergings were done by users expert in CVS and MolhadoRef.

Independent variables. Merging with MolhadoRef and merging with CVS.

Dependent variables. Time spent to perform the merging, the number of conflicts that cannot be solved automatically, the number of compile and runtime errors after merging.

5.3 Experimental Treatment

Now that we had real data about code developed in parallel, we wanted to merge implementations of ACK and MAX_HOPS into a code base that would contain both features. We used CVS text-based merging as the control group to test whether operation-based merging (with MolhadoRef) is more effective.

After we gathered all of the solutions implemented by the subjects, we created pairs as the cross product among the two groups of tasks (five solutions for task ACK, five solutions for task MAX_HOPS, resulting in 25 pairs). Each pair along with the base version of the LAN simulation forms a triplet. For each such triplet, we committed the source code in both CVS (using the Eclipse CVS client) and MolhadoRef. We first committed the base version, then checked it out in two different Eclipse projects, replaced the code in the checked out versions with the code for the MAX_HOPS and ACK tasks, then we committed the version containing the MAX_HOPS task (no merging was needed here), followed by committing the version containing the ACK task (merging was needed here).

TABLE 2
Effectiveness of Merging with CVS versus MolhadoRef

	Mean	Std.Dev.	Min.	Max
CVS Conflicts	8.04	2	4	11
MolhadoRef Conflicts	2.24	1.23	0	5
CVS Compile Errors	12.08	8.43	0	39
MolhadoRef Compile Errors	1.04	1.09	0	4
CVS Runtime Errors	0.75	1.11	0	5
MolhadoRef Runtime Errors	0.48	0.58	0	2
CVS Time to Merge[mins]	17.5	7.56	8	35
MolhadoRef Time to Merge[mins]	4.96	3.55	1	17

By not asking the subjects to do the merging, we prevented them from knowing the goal of our study so that they would not make subjective changes that could sabotage the outcome of merging. At the same time, we eliminated one of the independent variables that could affect the outcome of merging, namely their experience on merging with CVS or MolhadoRef. Instead, the first and second authors (who were both experts with CVS and MolhadoRef) did all of the mergings. To eliminate the memory effect, we randomized the order in which pairs were merged.

Table 2 shows the results of merging with Eclipse's CVS client versus MolhadoRef.

5.4 Statistical Results

After applying analysis of variance (ANOVA) using the Paired Student's t-test and Fisher's test, we were able to reject the null hypotheses and accept H1 (MolhadoRef automatically solves more conflicts), H2 (MolhadoRef produces fewer compile errors), and H4 (it takes less time to merge with MolhadoRef), at a significance level of $\alpha = 1\%$. We were not able to reject the null hypothesis for H3 (MolhadoRef produces fewer runtime errors) at $\alpha = 1\%$ level.

5.5 Threats to Validity

Construct validity. One could argue why we chose number of merge errors and time to merge as the indicators for the quality of merging. We believe that a software tool should increase the quality of the software and the productivity of the programmer. Compile and runtime errors both measure the quality of the merged code. The number of conflicting blocks indirectly measures how much of the tedious job is taken care of by the tool, while the time to merge directly measures the productivity of the programmer.

One could also ask why we did all of the merging ourselves instead of using the subjects. We wanted to avoid confusing the effect of the tool with the experience of the person operating the tool. We were experts with both CVS and MolhadoRef, whereas our subjects would not have any experience with MolhadoRef. In addition, the subjects did not know that their solutions would be merged. This way, we simulated an environment where the kinds of changes are not limited by whether or not they can be easily merged, but where programmers have absolute freedom to improve their designs.

Internal validity. One could ask whether the design of the experiment and the results truly represent a cause-and-effect relationship. For instance, since we were the only ones who merged subjects' solutions, the repetition of experiments could have influenced the results. To eliminate the memory effect, we randomized the order in which we merged pairs of solutions. In addition, we split the merging tasks into several clusters, separated by several days. Another question is whether the person who merged with MolhadoRef was better at merging than the person who merged with CVS. Before doing the merging experiment, we tried a few cold-run merging experiments and both persons involved in merging (the first and second authors) had the same productivity.

External validity. One could ask whether our results are applicable and generalizable to a wider range of software projects. We only used one single application and the input code developed in parallel was produced using Eclipse and Java. Maybe by using IDEs that do not feature refactorings, the programmers will make fewer refactorings. Although the presence of refactorings conveniently integrated within an IDE can affect the amount of refactoring, we noticed cases when subjects refactored manually. In addition, Java is a popular programming language and Eclipse is widely used to develop Java programs.

The subjects of our experiment were all graduate students. On one hand, this is an advantage because the subject demographics are relatively homogeneous. On the other hand, use of students limits our ability to generalize the results to professional developers. However, a careful look at Table 1 shows that the subjects had reasonable experience, most of them had worked in industry before coming to graduate school. Notably, two of them had several years of professional consulting and programming experience.

Reliability. The initial base version along with the students' solutions can be found online [17], so our results can be replicated.

6 Case Study

We also conducted a case study to further evaluate the effectiveness of MolhadoRef. We used MolhadoRef to merge its own source code. Most of the development of MolhadoRef was done by two programmers in a pair-programming fashion (two people at the same console). However, during the last three weeks, the two programmers ceased working on the same console. Instead, they worked in parallel; they refactored and edited the source code as before. When merging the changes with CVS, there were many same-line conflicts. It turned out that a large number of them were caused by two refactorings: One renamed a central API class `LightRefactoring` to `Operation`, while the other moved the API class `LightRefactoring` to a package that contained similar abstractions.

When merging the same changes using MolhadoRef, far fewer conflicts occur. Table 3 presents the effectiveness of merging with CVS versus MolhadoRef. Column "conflicts" shows how many of the changes could not be automatically merged and require human intervention. For CVS, these are changes to the same line or block of text. For MolhadoRef, these are operations that cannot be automatically incorporated in the merged version because they would have caused compile or runtime errors. The next columns show how many compile-time and runtime errors are introduced by each SCM.

TABLE 3
Effectiveness of Merging with CVS versus MolhadoRef

	MolhadoRef Case Study	LAN Simulation Example
CVS Conflicts	36	3
MolhadoRef Conflicts	1	1
CVS Compile Errors	41	1
MolhadoRef Compile Errors	0	0
CVS Runtime Errors	7	1
MolhadoRef Runtime Errors	0	0
CVS Time to Merge [mins]	105	5
MolhadoRef Time to Merge [mins]	2	1

The first rows show how many conflicts could not be solved automatically and required user intervention. The next rows show the number of compile and runtime errors after merging with each system. The last rows present the total time (including human and machine time) required to merge and then fix the merge errors.

Table 3 shows that MolhadoRef was able to automatically merge all 36 same-line conflicts reported by CVS. MolhadoRef asked for user assistance only once, namely when both developers introduced method `getID()` in the same class. MolhadoRef did not introduce any compile-time or runtime errors, while CVS had 48 such errors after "successful" merge. In addition, it took 105 minutes for the two developers to produce the final correct version using CVS, while it takes less than 2 minutes to merge with MolhadoRef.

7 Discussion

Our approach relies on the existence of logs of refactoring operations. However, logs are not always available. To exploit the full potential of record and replay of refactorings, we developed RefactoringCrawler [12] to automatically detect the refactorings used to create a new version. These inferred refactorings can be fed into MolhadoRef when recorded refactorings are not available.

Although one might expect that circular dependences would require a lot of manual editing, in practice such dependences are rare. Circular dependences can be eliminated manually by deleting some of the operations in the cycle or automatically by inverting refactorings (as seen in Fig. 6) or by the *enhanced* refactorings (Step #5). The enhanced refactorings eliminate dependences between refactorings that change the same program elements since these refactorings can be replayed in any order.

Our merging algorithm discriminates between refactorings and API edits. Although both of these operations have semantics that can be easily inferred by tools, MolhadoRef inverts and replays refactorings only, not API edits (although API edits' semantics are taken into account during conflict detection). Conceptually, API edits can be treated the same way in which MolhadoRef currently treats refactorings. There are two reasons why MolhadoRef treats them differently. First, API edits are harder to invert than refactorings since API edits are not behavior-preserving. Inverting an `AddAPIMethod` by removing the method would invalidate all of the edits that refer to the new method (e.g., method call sites). To fix this would require also inverting the code edits (e.g., removing the call sites that refer to new method). Second, API edits do not have the same global effect as the API refactorings because only one user (e.g., the one who introduced the new method) would be aware of the new API, leading to fewer cases of same-line conflicts than when refactorings are involved. Since there are far fewer benefits from inverting and replaying API edits, we decided to treat them like code edits.

MolhadoRef is built on top of the Molhado object-oriented SCM infrastructure [7], which was developed for creating SCM tools. Molhado is a database that keeps track of history. MolhadoRef translates Java source code (all Java 1.4 syntax is supported) into Molhado structures. At the time of check-in, MolhadoRef parses to the level of method and field declaration and creates a Molhado counterpart for each program element. The method/field bodies are stored as attributes of the corresponding declarations. For each entity, Molhado gives a unique identifier. When refactorings change different properties of the entities (e.g., names, method arguments), MolhadoRef updates the corresponding Molhado entries. Nevertheless, the identity of program entities remains intact even after refactoring operations (for a detailed description on implementation, see [14] and [10]).

Can such a refactoring-aware SCM system be implemented on top of a traditional SCM that lacks unique identifiers? We believe that, with enough engineering, the features of MolhadoRef can be retrofitted on a system like CVS. To retrieve the history of refactored elements, it is important to keep a record of unique identifiers associated with program elements. Identifier-to-name maps can be saved in metadata files and stored in the repository along with the other artifacts. At each check-out operation, the MolhadoRef CVS client needs to load these metadata files into memory so that they can be updated as the result of refactoring operations. At each check-in operation, these files are stored back into the repository.

Limitations of MolhadoRef. One obvious limitation is that our approach requires that the SCM be language-specific. However, we do not see this as a limitation but as a trade-off: We are intentionally giving up generality for gaining more power. This is no different from other tools used in software engineering, for example, IDEs are language-specific (along with all of the tools that make up an IDE, e.g., compilers, debuggers, and so forth), refactoring tools are all language-specific.

Second, adding support in MolhadoRef for a new kind of refactoring entails adding several cells in the conflict and dependence matrices, describing all combinations between the new operation and all existing operations. This can be a time-consuming task. We discovered that new cells tend to reuse predicates from earlier cells, which makes them easy to implement. Cells requiring new predicates are still time-consuming to implement. Third, the correctness of the system depends on the correctness of formulas in the conflict and dependence matrices. However, we carefully revisited those formulas. In addition, multiple experiments and more experience using our system can help empirically test the correctness of those formulas.

An alternative to the manual derivation of predicates is the analytical approach, like the one proposed by Kniesel and Koch [16]. However, this requires a formal model of refactoring preconditions. Since the current refactoring engines (including Eclipse's) lack such formalism, this entails implementing a whole new refactoring engine. We

chose instead to build upon a mature and thoroughly tested engine like Eclipse's.

8 Related Work

SCM systems have a long history [18], [19]. Early SCM systems (e.g., CVS [20]) provided versioning support for individual files and directories. In addition to version control, advanced SCM systems also provide more powerful configuration management services. Subversion [21] provides more powerful features such as versioning for metadata, properties of files, renamed or copied files/directories, and cheaper version branching. Similarly, commercial SCM tools still focus on *files* [19]. Advanced SCM systems also provide *fine-grained* versioning support not only for programs but also for other types of software artifacts. Examples include COOP/Orm [22], Coven [23], POEM [24], Westfechtel's system [25], Unified Extensional Versioning Model [26], Ohst et al.'s fine-grained SCM model [27], and so forth. However, none of them handle refactorings as MolhadoRef does.

Software merging. According to Mens [13], software merging techniques can be distinguished based on how software artifacts are represented. *Text-based* merge tools consider software artifacts merely as text (or binary) files. In RCS and CVS [20], lines of text are taken as indivisible units. Despite its popularity, this approach cannot handle two parallel modifications to the same line well. Only one of the two modifications can be selected, but they cannot be combined. Darcs [28] is a system rising in popularity, based on a unique algebra of patches. Darcs does not associate any semantics to a patch: A patch is just a series of textual changes. Darcs can only find out that two patches depend on each other if they affect the same portions of text. However, due to inheritance and method overriding in OO code, patches can affect each other, even when they are not lexically near each other.

Syntactical merging is more powerful than textual merging because it takes the syntax of software artifacts into account. Unimportant conflicts such as code comment or line breaks can be ignored by syntactic merger. Some syntactic merge tools focus on parse trees or abstract syntax tree [29], [30], [31]. Others are based on graphs [32], [33]. However, they cannot detect conflicts when the merged program is syntactically correct but semantically invalid. To deal with this, *semantic-based* merge algorithms were developed. In Wesfetchtel's context-sensitive merge tool [25], an AST is augmented by the bindings of identifiers to their declarations. More advanced semantic-based merge algorithms [34], [35], [36] detect behavioral conflicts using dependency graphs, program slicing, and denotational semantics.

Operation-based merging. The operation-based approach has been used in software merging [5], [32], [37], [38], [39]. It is a particular flavor of semantic-based merging that models changes between versions as explicit operations or transformations. The operation-based merge approach can improve conflict detection and allows better conflict solving [13]. Lippe and van Oosterom [5] describe a theoretical framework for conflict detection with respect to general transformations. No concrete application for refactorings was presented. Edwards' operation-based framework detects and resolves semantic conflicts from application-supplied semantics of operations [37]. GINA [40] used a *redo* mechanism to apply one developer's changes to the other developer's version. The approach has problems with long command histories and finer granularity of operations. The departure point of MolhadoRef from existing approaches is its ability to handle the merging of changes that involve both *refactoring* and *textual editing*.

Similarly to MolhadoRef, Ekman and Asklund [41] present a refactoring-aware versioning system. Their approach keeps the program elements and their IDs in volatile memory, thus allowing for a short-lived history of refactored program entities. In our approach, program elements and their IDs are modeled in the SCM and stored throughout the life cycle of the project, allowing for a global history tracking of refactored entities. In addition, their system does not support merging.

As described, fine-grained and ID-based versioning have been proposed before by others. However, the novelty of this work is the combination of semantic-based fine-grained ID-based SCM to handle refactorings and high-level edit operations. To the best of our knowledge, we are presenting the first algorithm to merge refactorings and edits. The algorithm is implemented and the first experiences are demonstrated.

9 Conclusions and Future Work

Refactoring tools have become popular because they allow programmers to safely make changes that can affect all parts of a system. However, such changes create problems for the current SCM tools that operate at the file level: Refactorings create more merge conflicts, the history of the refactored program elements is lost, and understanding of program evolution is harder.

We have presented a novel SCM system, MolhadoRef, that is aware of program entities and the refactoring operations that change them. MolhadoRef uses the operation-based approach to record (or detect) and replay changes. By intelligently treating the dependences between different operations, it merges edit and refactoring operations effectively. In addition, because MolhadoRef is aware of the semantics of change operations, a successful merge does not produce compile or runtime errors.

This research is part of our larger goal to upgrade component-based applications to use the latest version of a component by replaying the component refactorings [4], [12]. The upgrading tool needs to handle refactorings and edits not only on the component side but also on the application side. This is a special case of the more general merging case presented in this paper and, therefore, we will apply the same merge algorithm.

We believe that the availability of such semantics-aware refactoring-tolerant SCM tools will encourage programmers to be even bolder when refactoring. Without the fear that refactorings are causing conflicts with others' changes, software developers will have the freedom to make their designs easier to understand and reuse.

The reader can find screenshots and download MolhadoRef from its Web page [17].

ACKNOWLEDGMENTS

The authors would like to thank Darko Marinov, Don Batory, Frank Tip, John Brant, Jeff Overbey, Brett Daniel, Nicholas Chen, Mathieu Verbaere, Steve Berczuk, and the anonymous reviewers for their insightful comments on different drafts of this paper. D. Dig would also like to thank IBM for an Eclipse Innovation Grant and the University of Illinois Urbana-Champaign Computer Science Department for an Outstanding Mentoring Fellowship. This paper is an extended version of a conference paper [1].

REFERENCES

[1] D. Dig, K. Manzoor, R. Johnson, and T.N. Nguyen, "Refactoring-Aware Configuration Management for Object-Oriented Programs," *Proc. 29th Int'l Conf. Software Eng.*, pp. 427-436, 2007.
[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Adison-Wesley, 1999.
[3] S. Berczuk, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration,* private conversations with SCM consultant, Steve Berczuk, author, Addison-Wesley, 2002.
[4] D. Dig and R. Johnson, "How Do APIs Evolve? A Story of Refactoring," *J. Software Maintenance and Evolution,* vol. 18, no. 2, pp. 87-103, 2006.
[5] E. Lippe and N. van Oosterom, "Operation-Based Merging," *Proc. Fifth Symp. Software Development Environments,* pp. 78-87, 1992.
[6] Eclipse Foundation, http://eclipse.org, 2008.
[7] T.N. Nguyen, E.V. Munson, J.T. Boyland, and C. Thao, "An Infrastructure for Development of Object-Oriented, Multi-Level Configuration Management Services," *Proc. 27th Int'l Conf. Software Eng.*, pp. 215-224, 2005.
[8] W. Miller and E.W. Myers, "A File Comparison Program," *Software, Practice and Experience,* vol. 15, no. 11, pp. 1025-1040, 1985.
[9] S. Demeyer, F.V. Rysselberghe, T. Gîrba, J. Ratzinger, R. Marinescu, T. Mens, B.D. Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall, and M. El-Ramly, "The LAN-Simulation: A Refactoring Teaching Example," *Proc. Int'l Workshop Principles of Software Evolution,* pp. 123-134, 2005.
[10] D. Dig, T. Nguyen, and R. Johnson, "Refactoring-Aware Software Configuration Management," Technical Report UIUCDCS-R-2006-2710, Univ. of Illinois Urbana-Champaign, Apr. 2006.
[11] What's New in Eclipse 3.2 (JDT), http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jdt.doc.user/whatsNew/jdt_whatsnew.html, 2008.
[12] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automatic Detection of Refactorings in Evolving Components," *Proc. 20th European Conf. Object-Oriented Programming,* pp. 404-428, netfiles.uiuc.edu/dig/RefactoringCrawler, 2006.
[13] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Trans. Software Eng.,* vol. 28, no. 5, pp. 449-462, May 2002.
[14] D. Dig, K. Manzoor, R. Johnson, and T. Nguyen, "Refactoring-Aware Configuration Management System for Object-Oriented Programs," Technical Report UIUCDCS-R-2006-2770, Univ. of Illinois Urbana-Champaign, Sept. 2006.
[15] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms,* second ed., pp. 549-552. MIT Press and McGraw-Hill Books, 2001.
[16] G. Kniesel and H. Koch, "Static Composition of Refactorings," *Science of Computer Programming,* vol. 52, nos. 1-3, pp. 9-51, 2004.
[17] MolhadoRef Web page, https://netfiles.uiuc.edu/dig/MolhadoRef, 2004.
[18] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys,* vol. 30, no. 2, pp. 232-282, 1998.
[19] CM Yellow Pages, http://www.cmcrossroads.com/, 2008.
[20] T. Morse, "CVS," *Linux J.,* vol. 1996, no. 21es, p. 3, 1996.
[21] "Subversion," http://subversion.tigris.org/, 2008.
[22] B. Magnusson and U. Asklund, "Fine-Grained Revision Control of Configurations in COOP/Orm," *Proc. Sixth Software Configuration Management Workshop,* pp. 31-48, 1996.
[23] M.C. Chu-Carroll, J. Wright, and D. Shields, "Supporting Aggregation in Fine Grained Software Configuration Management," *Proc. Foundations of Software Eng.,* pp. 99-108, 2002.
[24] Y.-J. Lin and S.P. Reiss, "Configuration Management with Logical Structures," *Proc. 18th Int'l Conf. Software Eng.,* pp. 298-307, 1996.
[25] B. Westfetchtel, "Structure-Oriented Merging of Revisions of Software Documents," *Proc. Third Int'l Workshop Software Configuration Management,* pp. 68-79, 1991.
[26] U. Asklund, L. Bendix, H. Christensen, and B. Magnusson, "The Unified Extensional Versioning Model," *Proc. Ninth Software Configuration Management Workshop,* pp. 100-122, 1999.
[27] D. Ohst, M. Welle, and U. Kelter, "Differences between Versions of UML Diagrams," *Proc. Foundations of Software Eng.,* pp. 227-236, 2003.
[28] "Darcs SCM," http://darcs.net/, 2008.
[29] U. Asklund, "Identifying Conflicts during Structural Merge," *Proc. Nordic Workshop Programming Environment Research,* pp. 231-242, 1994.
[30] J.E. Grass, "CDIFF: A Syntax Directed Differencer for C++ Programs," *Proc. Usenix C++ Conf.,* pp. 181-194, 1992.
[31] W. Yang, "How to Merge Program Texts," *The J. Systems and Software,* vol. 27, no. 2, pp. 129-135, 1994.
[32] T. Mens, "A Formal Foundation for Object-Oriented Software Evolution," PhD dissertation, Vrije Univ. Brussels, 1999.
[33] J. Rho and C. Wu, "An Efficient Version Model of Software Diagrams," *Proc. Fifth Asia Pacific Software Eng. Conf.,* pp. 236-243, 1998.
[34] V. Berzins, "Software Merge: Semantics of Combining Changes to Programs," *ACM Trans. Programming Languages and Systems,* vol. 16, no. 6, pp. 1875-1903, 1994.
[35] J.W. Hunt and T.G. Szymanski, "A Fast Algorithm for Computing Longest Common Subsequences," *Comm. ACM,* vol. 20, no. 5, pp. 350-353, 1977.
[36] W. Yang, S. Horwitz, and T. Reps, "A Program Integration Algorithm that Accommodates Semantics-Preserving Transformations," *ACM Trans. Software Eng. Methodology,* vol. 1, no. 3, pp. 310-354, 1992.
[37] W. Edwards, "Flexible Conflict Detection and Management in Collaborative Applications," *Proc. 10th Ann. ACM Symp. User Interface Software and Technology,* pp. 139-148, 1997.
[38] A. Lie, R. Conradi, T.M. Didriksen, and E.-A. Karlsson, "Change Oriented Versioning in a Software Engineering Database," *Proc. Second Int'l Workshop Software Configuration Management,* pp. 56-65, 1989.
[39] H. Shen and C. Sun, "A Complete Textual Merging Algorithm for Software Configuration Management Systems," *Proc. 28th Ann. Int'l Computer Software and Applications Conf.,* pp. 293-298, 2004.
[40] T. Berlage and A. Genau, "A Framework for Shared Applications with a Replicated Architecture," *Proc. Sixth ACM Symp. User Interface Software and Technology,* pp. 249-257, 1993.
[41] T. Ekman and U. Asklund, "Refactoring-Aware Versioning in Eclipse," *Electronic Notes in Theoretical Computer Science,* vol. 107, pp. 57-69, 2004.

Danny Dig received the BS and MS degrees in computer science from the "Politehnica" University of Timisoara, Romania, where he built JavaRefactor (the first open-source refactoring engine for Java), and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 2007. He is a postdoctoral associate at the Massachusetts Institute of Technology. He is particularly interested in program transformations, automated refactoring, design and architectural patterns, and broadly interested in software reuse, software development processes, and software evolution. He is currently doing research on refactorings that increase the parallelism of existing sequential code. He served as the program and conference chair of the First Workshop on Refactoring Tools (2007).

Kashif Manzoor received the BS degree in computer systems engineering from Ghulam Ishaq Khan Institute of Engineering Sciences and Technology, Topi, Pakistan, in 1997 and the MS degree in computer science from the University of Illinois at Urbana-Champaign in 2007. He is currently a director of professional services at Techlogix. His research interests include refactoring, design patterns, process improvement, and software development methodologies.

Ralph Johnson received the BA degree from Knox College in 1977 and the PhD degree in computer science from Cornell University in 1987. He is a research associate professor at the University of Illinois at Urbana-Champaign. He is one of the four coauthors of *Design Patterns* and the leader of the group that developed the Smalltalk Refactoring Browser, the first refactoring tool. He is currently working on a tool for refactoring Fortran and for helping programmers tune their Fortran programs for new parallel architectures. He is a member of the IEEE Computer Society.

Tien N. Nguyen received the PhD degree in computer science from the University of Wisconsin in 2005. He is an assistant professor in the Electrical and Computer Engineering Department at Iowa State University. His software engineering expertise is in the areas of version control and configuration management, software maintenance and evolution, and program analysis. His research work on semantics-based configuration management and collaborative supports has produced Molhado, the first object-oriented configuration management infrastructure, which has been successfully used in several projects. He is a member of the IEEE.