Refactoring-aware Configuration Management for Object-Oriented Programs

Danny Dig, Kashif Manzoor, Ralph Johnson
University of Illinois at Urbana-Champaign
{dig, manzoor2, rjohnson}@uiuc.edu

Tien N. Nguyen
Iowa State University
tien@iastate.edu

Abstract

Current text based Software Configuration Management (SCM) systems have trouble with refactorings. Refactorings result in global changes and lead to merge conflicts. A refactoring-aware SCM system reduces merge conflicts, preserves program history better and makes it easier to understand program evolution. 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 shows that MolhadoRef solves automatically more merge conflicts than CVS while resulting in fewer merge errors.

1. Introduction

Refactorings are program transformations that improve the internal design without changing the observable behavior [15] (e.g., renamings, moving methods between classes, changing method signatures). Automated refactoring tools have become very popular because they allow programmers to change source code quicker and safer than manually. However, refactoring tools make particular demands on text-based SCM systems.

Traditional SCM systems work best with modular systems. Different programmers tend to work on different modules and so it is easy to merge changes. But refactorings cut across module boundaries and cause changes to many parts of the systems. 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.

The state-of-practice process for refactoring on large projects is for all developers to check in their code before they leave for the weekend. The senior designer then makes these global changes (e.g., API changes) and commits the refactored code. Upon their return, developers check out the refactored versions. However, this serializes the development of code. In addition, by forcing refactorings to be performed only by 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 [9] 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% 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. Since they signal merge conflicts only when two users change the same line of code, even a successful merge might result in an incorrect program. 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 that works for Java, and the merge algorithms at its core. MolhadoRef has several important advantages over a traditional text-based SCM:

1. Better merging. 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 the refactoring operations.

2. Better preservation of program history. MolhadoRef tracks the history of refactored program elements even when they are renamed or moved to different files (e.g., when moving a method to a different class).

3. Better understanding of program evolution. Some refactoring operations (like renaming a popular public

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

method) may cause thousands of changes (e.g., updating all call sites) scattered throughout the code. By displaying the evolution of code in terms of higher-level operations (e.g., refactorings), MolhadoRef hides the complexity caused by the sheer amount of low-level changes corresponding 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 [20], in other words it treats both refactorings and edits as change operations that are recorded and replayed. If all edits came before refactorings, it would be easy to merge the two versions by first doing a three-way merging then replaying the refactorings. But edits and refactorings are mixed, so we have to *invert* refactorings to commute an edit and a refactoring. Moreover, refactorings will sometimes have *dependences* between them.

MolhadoRef uses Eclipse as the front-end for changing code and customizes Molhado [26], a framework for SCM, to store Java programs. Although the merging algorithm is independent of the Molhado infrastructure and can be reused with other SCM backends, building on top of an ID-based SCM like Molhado allows our system to keep track of the refactored entities. When evaluating MolhadoRef on three weeks of its own development, 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 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:

- without losing any power to merge manual edits, it converts refactorings from being the weakest link in an SCM system to being the strongest
- it presents the first algorithm to effectively merge refactorings and edit operations
- it describes the implementation of the algorithm and evaluates the effectiveness of a refactoring-aware SCM system on real world software

2. Motivating Example

To see the limitations of text-based SCM, consider the simulation of a Local Area Network (LAN) used as a refactoring teaching example [7] in many european universities (shown in Figure 1).

Initially, there are five classes: `Packet`, a superclass `LANNode` and its subclasses `PrintServer`, `NetworkTester`, and `Workstation`. All `LANNode` objects are linked to each other 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, 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 `nodes.PrintServer` to `content.Packet` (τ_1). Next, she defines a new method, `sendPacket(Packet)`, in class NetworkTester (τ_2). 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). Lastly, 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). Lastly, 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 signals conflicts only 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), a textual merging could not successfully merge these changes. This is an unnecessary merge conflict because a tool that understood the semantics of the changes could 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). Though tedious to fix, such an error is easy to catch.

Other errors result in programs that compile but have unintended changes to their behavior. For instance, because Alice introduced a new method `sendPacket` in subclass `NetworkTester` and Bob renames the polymorphic

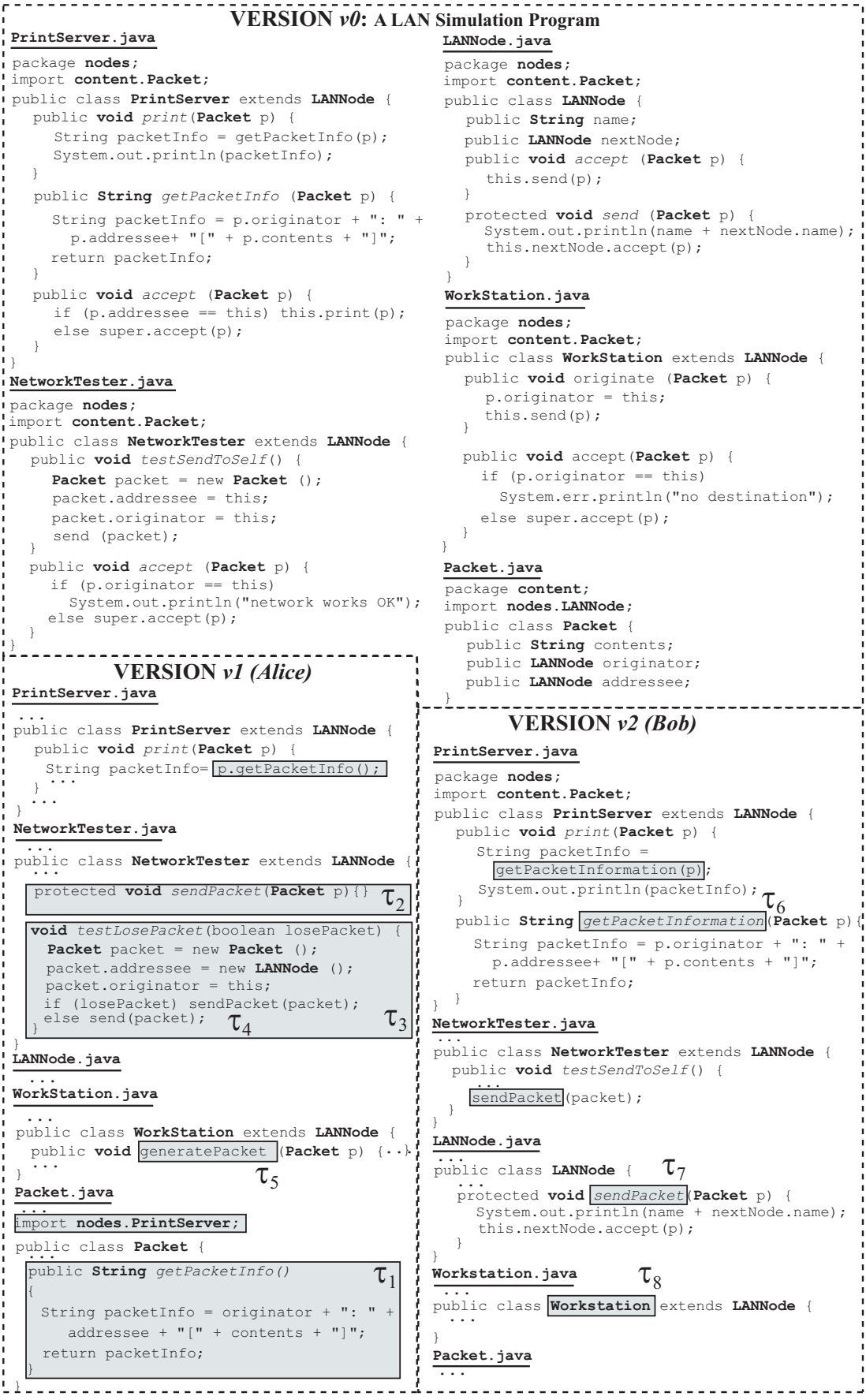

Figure 1. Motivating Example. Boxes show the changes in each version.

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

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 originally this method call used the implementation of `LANNode.send`. Since this type of conflict is not reported during compilation, the merged program contains bugs that require many hours to find.

Current SCM systems lose the history of refactored program entities. For instance, once Alice moved method `getPacketInfo(Packet)` from class `PrintServer` to `Packet`, the history of the `getPacketInfo` is effectively lost since a file-based SCM repository maintains the method as if it is a newly defined method in the class `Packet`. Thus, a file-based SCM tool could not help a developer understand code evolution when program entities are refactored. .

Because there were so many refactorings in version V_1 and V_2, it is hard for another developer who only saw version V_0 to understand the evolution of the code. A file-based SCM is of less use in program understanding because it floods the developer with all the low level changes that happened between V_0 and V_2.

3. Background and Terminology

Our approach to refactorings-tolerant SCM systems is based on a different paradigm, called *operation-based* [20]. 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 a correct usage of all the APIs. For this reason, we distinguish between operations that affect the APIs and those that do not. MolhadoRef treats a version as 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 [9]). MolhadoRef handles the following *API edits*: added package, deleted package, added class, 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. But refactorings are the operations with the most well defined semantics, so the ones that can benefit the most. 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: $\tau : Program -> Program$

When necessary, we make the distinction between refactorings, represented with ρ and edits, represented with 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 the new name is not in use. Applying an operation τ_i inappropriately to a program P results in an invalid program, represented by $\perp$. The result of applying an operation to $\perp$ is $\perp$.

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

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 any 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 any order produces an invalid program:
$\tau_j; \tau_i(P) = \perp \wedge \tau_i; \tau_j(P) = \perp$

For instance, 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 run-time errors. These conflicts always 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 τ_i ($\tau_i \prec_P \tau_j$) if τ_j and τ_i do not commute:
$\tau_i \prec_P \tau_j$ iff $\tau_i; \tau_j(P) \neq \perp \wedge (\tau_i; \tau_j(P) \neq \tau_j; \tau_i(P))$

The $\prec_P$ dependence is *strict* partial order, that is, it is irreflexive, antisymmetric 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, scenarios in which the names of program entities change would never pose a problem [10]. To make name-based refactoring engines be ID-based requires rewriting the whole 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 rename class `WorkStation`). The second is to modify the refactoring engine so that in addition to changing source code, it also changes subsequent refactorings (e.g., during the replay of renaming class `WorkStation` to `Workstation`, the refactoring engine changes the representation of rename method refactoring RenM(WorkStation.originate, WorkStation.generatePacket) to RenM(Workstation.originate, Workstation.generatePacket)). Our merging algorithm uses both approaches.

4. Merging Algorithm

4.1. High level overview

We illustrate the merging algorithm (see Fig. 2) using the LAN simulation example presented earlier. 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.

Step #1 detects the API and code edits through 3-way differencing between V_1, V_2 and V_0. In V_1 it detects two added methods, τ_2 and τ_3, in V_2 it detects none.

Step #2 searches for compile and run-time conflicts in API edits and refactorings. It detects a conflict between τ_2 and the rename method refactoring, τ_7. 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 he choses `losePacket` instead of `sendPacket`). The algorithm also searches for possible circular dependences between refactorings. If any are found, the user deletes one of the refactorings involved in cycle (in this example no circular dependence exist). This process of detecting/solving continues until no more conflicts or circular dependences remain.

```
INPUT = {V_2, V_1, V_0, refLog_2, refLog_1}
Operations refs= refLog_1 + refLog_2
Operations edits= detectEdits(V_2,V_1,V_0,refs) #1
repeat{              #2
  {edits, refs}= userSolvesConflicts({edits, refs})
  Graph refsDAG = createRefDependenceGraph(refs)
  {refs, refsDAG} =
   userEliminatesCircularDependences(refs, refsDAG)
} until noConflicsOrCircDependences(edits, refsDAG)
Version V_1_minusRef= invertRefactorings(V_1, refs) #3
Version V_2_minusRef= invertRefactorings(V_2, refs)
Version V_merged_minusRef=                      #4
   3-wayTextualMerge(V_2_minusRef, V_1_minusRef, V_0)
orderedRefs= topologicalSort(refsDAG)   #5
Version V_merged=
replayRefactorings(V_merged_minusRef, orderedRefs);
OUTPUT = {V_merged}
```

Figure 2. Overview of the merging algorithm

Step #3 inverts each refactoring in V_1 and V_2 by applying another refactoring. For instance, it inverts τ_1 by moving method `getPacketInfo` back to `PrintServer`, and it inverts τ_8 by renaming `Workstation` back to `WorkStation`. By inverting refactorings, all the edits that were referencing the refactored program entities are changed to refer to the old version of the entities. This step produces two software components that contain all the changes in V_1, respectively V_2, except refactorings.

Step #4 merges textually (using a modified version of the three-way merging [24]) all the API and code edits from V_1^{-Ref} and V_2^{-Ref}. Since the refactorings were previously inverted, all same-line conflicts that would have been caused by refactorings are eliminated. For instance, 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_{merged}^{-Ref}.

Step #5 replays on V_{merged}^{-Ref} the refactorings that happened in V_1 and V_2. Before replaying, the algorithm reorders all the refactorings using the dependence relations. Replaying the refactorings incorporates their changes into the V_{merged}^{-Ref} which already contains all the edits. For instance, replaying a method renaming updates all the call sites to that method that were introduced as edits.

4.2. Detecting Operations

Refactorings in versions V_2 and V_1 are recorded at the time when they were performed by Eclipse's [12] new refactoring engine. If Eclipse did not have this record feature, refactorings could be inferred using RefactoringCrawler [8].

To detect the API edits and code edits, the algorithm employs a three-way textual *differencer* (since two-way differencer cannot distinguish between additions and dele-

tions [23]). This differencer detects lines, files, and folders that were changed. From this low level information, the algorithm constructs the higher level, API edits. Out of these edits, the algorithm removes those created by refactorings.

Even though the scope of our merging is at the API level, to correctly signal compile- or run-time conflicts, the algorithm detects a few code edits that are below the API level. These include add and delete method calls. For instance, if Alice deletes the method declaration `accept` and Bob adds a method call to `accept`, this results in a compile conflict.

4.3. Detection and Solving of Conflicts and Circular Dependences

MolhadoRef detects conflicts between operations by using a matrix of predicates. For any two kinds of operations, the matrix gives a predicate that indicates whether the operations conflict. This matrix includes refactorings, API edits, and the code edits that are tracked.

For example, suppose τ_i is RenameMethod(m_1, m_2) and τ_j is RenameMethod(m_3, m_4). These two renamings result in a conflict if (i) the source of both refactorings is the same (e.g., $m_1 = m_3$), or (ii) the destination of both refactoring is the same (e.g., $m_2 = m_4$). Due to polymorphic overriding, we must also consider the case when the source methods are not in the same class, but one overrides the other.

When the source of both refactorings are the same (i), 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 run-time conflict because the initial overriding relationship would be broken. When the destination of the two refactorings is the same (ii), 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 run-time conflict because of accidental method overriding.

FOPL formulae describing all possible combinations of operations (both refactorings and edits) detected in step #1 are in a companion tech report [11].

Circular Dependences. When there is an ordering dependence between two operations, the algorithm choses the correct order in which to replay the operations. Initially, there is a total order (or linear order) of the change operations in each version, given by the time sequence in which these operations were applied. However, operations can be replayed in any order, unless there is a dependence between them, so that 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. When $\tau_i \prec_P \tau_j$, the algorithm adds a directed edge from τ_i to τ_j. Next, the algorithm searches for cyclic dependences. There can only be cycles between operations from two users, not between operations from the same user because for each user it was initially possible to play all the operations. After it finds all cycles, it presents them to the user who must choose how to eliminate cycles. Although one expects that circular dependences would require the user to do a lot of manual editing, in practice such dependences occur very seldom. Assuming that there are no more cycles, all operations are in a directed acyclic graph (DAG).

User-assisted Conflict Resolution. Circular dependences and compile and run-time 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. To solve the conflicts caused by name collision, the user must select a different name.

Once the users solves the conflicts, the algorithm searches again for conflicts. This sequence of finding conflicts and solving conflicts is repeated until there are no more conflicts. The algorithm always converges to a fixed point. Informally, this happens because during manual conflict resolution the user keeps deleting conflicting operations, thus the total number of change operations keeps decreasing (in the worst case he can keep deleting change operations until there are no more operations).

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.

There is an important distinction between what we mean by inverting a refactoring and how the popular refactoring engines (like Smalltalk RefactoringBrowser, Eclipse or IntellijIdea) undo a refactoring. To decrease memory usage and avoid recomputations, 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.

This approach is not suitable for MolhadoRef. The only way to undo a refactoring is to first undo all the operations that come after it. 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).

Besides overcoming the drawbacks discussed above, another important benefit is that inverting a refactoring also changes the edits. Recall the motivation ex-

ample where Bob renamed method `getPacketInfo` to `getPacketInformation` and later he added 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 updated too, 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.

Just as refactorings have preconditions, inverting a refactoring has preconditions too, and if those preconditions are not met then a refactoring cannot be inverted. We have some heuristics that handle such cases by adding program transformations or storing additional information before inverting a refactoring. For instance, if Bob renames `PrintServer.getPacketInfo` to `getPacketInformation` and then adds a new method in the same class called `getPacketInfo`, inverting the rename refactoring is not possible because of the newly introduced method. The algorithm searches for potential name collisions before inverting the refactoring, and executes another refactoring to avoid the collision. In this case, the algorithm gives the newly introduced `getPacketInfo` a unique name, and tags this rename refactoring. In step #5, after all the regular refactorings have been replayed, the algorithm inverts all refactorings marked with tags.

Consider the case when Bob changes the signature of a method `sendPacket` by adding an extra argument of type integer with a default value 0 to be used in method calls, and later he adds a method call where he passes value 7. Inverting the refactoring and redoing it naively would lose the value 7 and replace it with value 0. Before inverting the refactoring, the merge algorithm 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.

When no heuristic for inverting a refactoring is found, the algorithm treats the refactoring as a textual edit: 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 proceed and in the worst case it is as good as classic textual merging. The heuristics are good enough to invert all the refactorings in the case studies.

4.5. Textual Merging

Once refactorings are inverted, all 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 [24] 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. For instance, although both users changed the declaration of `getPacketInfo`, the call to this method inside `PrintServer.print` no longer causes same-line conflict.

Still, if two users change the same lines by code edits, this generates a same-line conflict. If Alice and Bob change the same lines in a file by API edits (e.g., each adds a new method declaration), MolhadoRef merges this automatically using the semantics of API edits. In general, when same lines are changed by edits, our algorithm is slightly better than traditional text-based merging. 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 a refactoring cannot use a name unless it is current. But when refactorings are merged from two different streams, there are two ways that renaming can interfere.

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 to a method are performed before the refactorings that rename its class. More precisely, MolhadoRef uses a topological sort algorithm [6] to reorder the nodes in the refactorings DAG.

The second is where the 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 will be 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 (τ_6, τ_1). Changing either the method name or the class name will invalidate the other refactoring. Mohaldoref 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.

5. Implementation

Programming tools are more likely to be used when they are conveniently incorporated in an Integrated Devel-

opment Environment (IDE) such as Eclipse. We implemented an operation-based SCM as an Eclipse plugin, MolhadoRef. MolhadoRef uses the Eclipse Java programming editor as the front end, and customizes Molhado framework [26] to store Java programs. Molhado is an object-oriented SCM infrastructure, which was developed for creating SCM tools. Essentially, Molhado is a database that keeps track of history.

MolhadoRef translates Java source code (all Java 1.4 syntax is supported) into Molhado structure. At the time of check-in, it parses to the level of method and field declaration and creates a Molhado counterpart for each program element that it parses. 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.

After code is checked in for the first time, subsequent 'check-in's need to store only the changes from last check-in. In a pure operation-based SCM, all the changes are recorded when they happen and are stored as operations in the SCM system. These operations are then replayed on the source code of a user who wants to update to the latest version. This operation-based approach can be very accurate in recording the exact type of change, but uses a large number of change operations and so recording and replaying can be slow. In contrast, the state-based approach computes deltas just before the user commits the code by comparing the two versions. This is more efficient (since the changes are computed only once per programming session) but it cannot recover the semantics of the changes (it detects all changes in a large pile of seemingly unrelated changes). For instance, a method rename can result in a lot of changes: changing the declaration of the method, updating the method callers as well as the transitive closure of all declarations and call sites of overridden methods.

MolhadoRef uses a mixture of both paradigms to maximize efficiency and accuracy. MolhadoRef uses the Eclipse compare engine to learn the individual deltas (e.g., changes within a method body or addition/removal of classes, methods, and fields) and it captures the refactorings performed by the Eclipse refactoring engine to record the semantics of refactoring operations.

Eclipse refactorings are based on a processor-participant architecture. We implemented participants for all the refactorings in Section 3. Our participants change the subsequent refactorings in a chain. More precisely, our participants update the fully qualified names of program elements that appear in the descriptors of subsequent refactorings.

Can MolhadoRef be implemented on top of a traditional SCM that lacks unique identifiers? We believe that the qualities of MolhadoRef can be retrofitted on a system like CVS. For the purpose of being able to retrieve history of refactored elements, it is important to keep a record of their unique identifiers. Identifier-to-name maps can be saved in metadata files and stored in the repository. At each check-out operation, the MolhadoRef CVS client needs to load these metadata files into memory so that they can be updated during refactoring. At each check-in operation, these files are stored back into the repository.

Although MolhadoRef currently handles a subset of the refactorings described in refactorings catalogs (e.g., [15]), a glance over the remaining refactorings in [15] shows that MolhadoRef can be further extended to support other refactorings. The key property we require of refactorings is the ability to invert refactorings.

6. Case study

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 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 analyze the parallel development of MolhadoRef itself.

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, much fewer conflicts occur. Table 1 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 run-time errors. Next columns show how many compile-time and run-time errors are introduced by each SCM.

Table 1 shows that MolhadoRef was able to automat-

	CVS			MolhadoRef		
CaseStudy	Conflicts	CompileErrs	RuntimeErrs	Conflicts	CompileErrs	RuntimeErrs
MolhadoRef code	36	41	7	1	0	0
MotivatingExample	3	1	1	1	0	0

Table 1. Effectiveness of merging with CVS versus MolhadoRef

ically 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 run-time 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 one minute for MolhadoRef.

Second, MolhadoRef helps in program understanding by reducing the complexity of all the low level textual changes. MolhadoRef raises the granularity level of changes from textual changes to structural changes. During the last 12 weeks of MolhadoRef development, there were 67 refactorings which correspond to 1267 changed lines in MolhadoRef and its accompanying JUnit test suite. Undoubtedly, it is easier to read and understand 67 changes than 1267 (a reduction of 1 : 19).

Third, being an ID-based SCM, MolhadoRef can retrieve the history of refactored program entities. For the three weeks of MolhadoRef development that we analyzed, CVS lost the history of two core files containing 73 API methods.

7. Related Work

SCM systems have a long history [5, 34]. Traditional systems (e.g. CVS [25]) provided versioning support for individual files and directories. In addition to version control, advanced SCM systems also provide more powerful configuration management services. Subversion [30] provides more powerful features such as versioning for metadata, properties of files, renamed or copied files/directories. Similarly, commercial SCM tools still focus on *files* [34]. Advanced SCM systems provide *fine-grained* versioning support not only for programs but also for other types of software artifacts. Examples include COOP/Orm [21], Coven [4], POEM [19], Westfechtel's system [31], and Ohst's fine-grained SCM model [27]. However, none of them handle refactorings as MolhadoRef does.

Software Merging. According to Mens [23], 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 [25], lines of text are taken as indivisible units. Despite its popularity, this approach cannot handle well two parallel modifications to the same line. Only one of the two modifications can be selected, but they cannot be combined. *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 [1, 16, 32]. Other are based on graphs [22, 28]. 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 [31], an AST is augmented by the bindings of identifiers to their declarations. More advanced semantic-based merge algorithms [3, 17, 33] detect behavioral conflicts using dependency graphs, program slicing, and denotational semantics.

Operation-based Merging. The operation-based approach has been used in software merging [13, 18, 20, 22, 29]. It is a particular flavor of semantic-based merging that models changes between versions as explicit operations. Operation-based merge approach can improve conflict detection and allows better conflict solving [23]. Lippe *et al.* [20] describes 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 [13]. GINA [2] used a *redo* mechanism to apply one developer's changes to other's version. The approach cannot handle well long command histories and the fine granularity. The departure point of MolhadoRef from existing approaches is its ability to merge *refactorings* and *edits*.

Similar to MolhadoRef, Ekman and Asklund [14] 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 stored throughout the lifecycle of the project, allowing for a global history tracking of refactored entities. Also, they do not describe any merging algorithm.

Fine-grained and ID-based versioning have been proposed before by others. However, the novelty of this work is the combination of operation-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.

8. 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 present 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. Storing the IDs of program entities across versions tracks the history better, while explicit representation of refactorings reduces the load of understanding the program evolution.

Because MolhadoRef is integrated with a popular development environment like Eclipse, we expect to have a large customer base. Future work will evaluate empirically the productivity of a group that uses MolhadoRef.

This research is part of our larger goal to upgrade component-based applications to use the latest version of component by replaying the component' refactorings [9, 8]. The upgrading tool needs to handle refactorings and edits not only on the component side, but on the application side too. 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 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 screen shots and download MolhadoRef at: http://netfiles.uiuc.edu/dig/MolhadoRef

References

[1] U. Asklund. Identifying conflicts during structural merge. In *Proceedings of Nordic Workshop on Programming Environment Research*, pages 231–242, 1994.

[2] T. Berlage and A. Genau. A framework for shared applications with a replicated architecture. In *UIST '93*, pages 249–257. ACM Press, 1993.

[3] V. Berzins. Software merge: semantics of combining changes to programs. *ACM Trans. Program. Lang. Syst.*, 16(6):1875–1903, 1994.

[4] M. C. Chu-Carroll, J. Wright, and D. Shields. Supporting aggregation in fine grained software configuration management. In *FSE'02*, pages 99–108. ACM Press, 2002.

[5] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*, pages 549–552. The MIT Press and McGraw-Hill Book Company, 2001.

[7] S. Demeyer, F. 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. In *IWPSE'05*, pages 123–134, 2005.

[8] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. In *ECOOP'06*, pages 404–428, 2006.

[9] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of Soft. Maintenance and Evolution*, 18:87–103, 2006.

[10] D. Dig, T.N. Nguyen, and R. Johnson. Refactoring-aware software configuration management. Technical Report UIUCDCS-R-2006-2710, UIUC, April 2006.

[11] D. Dig, K. Manzoor,R. Johnson, and T.N. Nguyen. Refactoring-aware Configuration Management System for Object-Oriented Programs. Tech. Report UIUCDCS-R-2006-2770, UIUC, Sept 2006.

[12] 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.

[13] W. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. In *Proceedings of UIST*, 1997.

[14] T. Ekman and U. Asklund. Refactoring-aware versioning in eclipse. *Electr. Notes Theor. Comput. Sci.*, 107:57–69, 2004.

[15] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Adison-Wesley

[16] J. Grass. A Syntax Directed Differencer for C++ Programs. In *In Proceedings of the Usenix C++ Conference*, 1992.

[17] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[18] A. Lie, R. Conradi, T. M. Didriksen, and E.-A. Karlsson. Change oriented versioning in a software engineering database. In *SCM-2*, pages 56–65. ACM Press, 1989.

[19] Y. Lin and S. Reiss. Configuration management with logical structures. In *ICSE'96*, pages 298–307, 1996.

[20] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE-5*, pages 78–87. ACM Press, 1992.

[21] B. Magnusson and U. Asklund. Fine-grained revision control of Configurations in COOP/Orm. In *SCM-6*. Springer Verlag, 1996.

[22] T. Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Vrije Universiteit Brussel, 1999.

[23] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. on Software Engineering*, 28(5):449–462, 2002.

[24] W. Miller and E. W. Myers. A file comparison program. *Softw., Pract. Exper.*, 15(11):1025–1040, 1985.

[25] T. Morse. CVS. *Linux Journal*, 1996(21es):3, 1996.

[26] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An Infrastructure for Development of Multi-level, Object-oriented Configuration Management Services. In *ICSE'05*, pages 215–224. ACM Press, 2005.

[27] D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. In *FSE'03*, pages 227–236, 2003.

[28] J. Rho and C. Wu. An efficient version model of software diagrams. In *APSEC'98*, page 236, 1998.

[29] H. Shen and C. Sun. A complete textual merging algorithm for software configuration management systems. *Proceedings of COMPSAC'04*, 01:293–298, 2004.

[30] Subversion.tigris.org. http://subversion.tigris.org/.

[31] B. Westfetchtel. Structure-oriented Merging of Revisions of Software Documents. In *SCM-3*, pages 68–79. Springer Verlag, 1991.

[32] W. Yang. How to merge program texts. *The Journal of Systems and Software*, 27(2), November 1994.

[33] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Trans. Softw. Eng. Methodol.*, 1(3):310–354, 1992.

[34] CM Yellow Pages. http://www.cmcrossroads.com/.