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.

1. Introduction

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 [11] 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 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 [21], 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 dependencies between them.

MolhadoRef uses Eclipse as the front-end for changing code and customizes Molhado [27], 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 more safely and more automatically than CVS while never losing the history of refactored entities. 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 [8] 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 (\( \tau_1 \)). Next, she defines a new method, sendPacket(Packet), in class Network-Tester (\( \tau_2 \)). 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 (\( \tau_3 \)) and implements it to call method sendPacket (\( \tau_4 \)). Lastly, Alice renames WorkStation.originiate(Packet) to generatePacket(Packet) (\( \tau_5 \)). 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) (\( \tau_6 \)). He also renames the polymorphic method LANNode.send() to sendPacket (\( \tau_7 \)). Lastly, Bob renames class WorkStation to Workstation (different capitalization \( \tau_8 \)). 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 (\( \tau_1 \)) while Bob altered the declaration location of the same method through renaming (\( \tau_6 \)), 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 (\( \tau_7 \)). 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 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 (\( \tau_2 \)) to simulate loss of packets, while originally this method called 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.
Figure 1. Motivating Example. Boxes show the changes in each version.
Thus, a file-based SCM tool could not help a developer understand code evolution when program entities are refactored. For example, it could not tell that the method `getPacketInfo` of the class `Packet` has originated from the method `getPacketInfo` of the class `PrintServer`.

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 [21]. 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 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 [10, 11]). 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: $\tau : Program \rightarrow Program$.

When necessary, we make the distinction between refactorings, represented with $\rho$ and edits, represented with $\sigma$. 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 $\tau_i$ inappropriately to a program $P$ results in an invalid program, represented by $\perp$. The result of applying an operation to $\perp$ is $\perp$.

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

The application of two operations is modeled by the function composition, denoted by “;”. ; also models the precedence: $\tau_i; \tau_j$ means first apply $\tau_i$ and then apply $\tau_j$ 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 conflicts that produce runtime errors. These conflicts always involve method overriding, such as the accidental method overriding between $\tau_2$ and $\tau_7$.

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:** $\tau_j$ depends on $\tau_i \prec_P \tau_j$ if $\tau_j$ and $\tau_i$ do not commute: $\tau_i \prec_P \tau_j \Leftrightarrow \tau_i; \tau_j(P) \neq \perp \wedge (\tau_i; \tau_j(P) \neq \tau_j; \tau_i(P))$

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

Dependences can exist between operations performed by the same user or between operations performed by the two users. For example, $\tau_4$ and $\tau_2$, operations performed by the same user, have a dependence that could produce an invalid program. Editing a new method call $(\tau_2)$ to `sendPacket` is dependent upon first adding the method declaration $(\tau_8)$ to which the method call binds, thus $\tau_8 \prec_P \tau_2$.

An example of dependence is the renaming of method `WorkStation.originate` to `generatePacket()` done by Alice $(\tau_5)$ and the renaming of class `WorkStation` to `Workstation` done by Bob $(\tau_8)$. If $\tau_5$ is played first, the replaying of $\tau_8$ is not possible because at this time the fully qualified name `WorkStation.originate` no longer exists, thus $\tau_5 \not\prec_P \tau_8$. 
This dependence between \( \tau_3 \) and \( \tau_8 \) exists because current refactoring engines are based on the names of the program entities, and class WorkStation no longer exists after replaying \( \tau_8 \). 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 [12]. 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 modifies all the refactorings. This is unfeasible, so the next best solution is to emulate ID-based engines.

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.

\[
\text{INPUT} = \{V_2, V_1, V_0, \text{refLogs}\}
\]

Operations op= 3-wayComparison(V_2,V_1,V_0) #1
Operations edits= detectEdits(op, refs) #2
repeat{
  \{edits, refs\} = userSolvesConflicts(\{edits, refs\})
  Graph refsDAG = createRefDependenceGraph(refs)
  \{refs, refsDAG\} =
    userEliminatesCircularDependences(refs, refsDAG)
  until noConflictsOrCircDependences(refs, refsDAG)
Version V_1_minusRef= invertRefactorings(V_1, refs) #3
Version V_2_minusRef= invertRefactorings(V_2, refs)
Version V_merged_minusRef= 3-wayTextualMerge(V_2_minusRef, V_1_minusRef, V_0) #4
orderedRefs= topologicalSort(refsDAG) #5
Version V_merged= replayRefactorings(V_merged_minusRef, orderedRefs);
OUTPUT = \{V_merged\}

Figure 2. Overview of the merging algorithm

Step #1 detects the API edits through 3-way differencing between \( V_1 \), \( V_2 \) and \( V_0 \). In \( V_1 \) it detects two added methods, \( \tau_2 \) and \( \tau_8 \) 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 \( \tau_2 \) and the rename method refactoring, \( \tau_7 \). 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.

Step #3 inverts each refactoring in \( V_1 \) and \( V_2 \) by applying another refactoring. For instance, it inverts \( \tau_7 \) by moving method getPacketInfo back to PrintServer, and it inverts \( \tau_8 \) 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 [25]) 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

To detect refactorings, API edits and code edits, the algorithm analyzes the three versions \( V_0 \), \( V_1 \), \( V_2 \). Recent extensions to refactoring engines (e.g., [13]) 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 plugin, 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 a tool for inferring refactorings, RefactoringCrawler [9].
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 deletions [24]). This differencer detects lines, files, and folders that were changed. From this low level information, the algorithm constructs the higher level, semantic API edits.

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 edit operations that are below the API level. These include add/delete method call, add/delete class instantiation, add/delete class inheritance, add/delete typecast. For instance, if Alice deletes the method declaration accept and Bob adds a method call to accept, this results in a compile conflict.

Some of the edit operations overlap with or are the side effects of refactorings. For instance, after renaming class WorkStation to Workstation, it appears as if WorkStation was deleted and Workstation was added. The algorithm discards these two change operations since they are replaced by the higher level refactoration operation. Other times, API refactorings are not tracked because they are replaced by edit operations (for instance, extract method into another public method is replaced by add method declaration).

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 \( \tau_i \) is RenameMethod\( (m_1, m_2) \) and \( \tau_j \) 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 refactorings 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.

More formally, using first-order predicate logic (FOPL):

\[
\text{hasConflicts(RenM}(m_1, m_2), \text{RenM}(m_3, m_4)) : \\
\left( m_1 = m_3 \lor \text{overrides}(m_1, m_3) \right) \land \\
\left( \text{simpleName}(m_2) \land \text{overrides}(m_2, m_4) \right) \\
\lor \\
\left( m_1 \land \neg \text{overrides}(m_1, m_3) \right) \land \left( m_2 \lor \neg \text{overrides}(m_2, m_4) \right)
\]

Similar FOPL formulae describing all possible combinations of operations (both refactorings and edits) detected in step #1 are in the appendix.

Circular Dependences. When there is an ordering dependence between two operations, the algorithm chooses 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 \( \tau_i \) to \( \tau_j \). 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. 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 syntactic or semantic conflicts caused by name
The new call site to getPacketInformation naming getPacketInformation rename method refactoring with the inverse refactoring (replacing the call site altogether would have introduced a different conflict, 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 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 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 [25] 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 (not refactorings), this generates a same-line conflict. If Alice and Bob change the same lines in a file by API edits...
The presence of \( \theta \) transformations elegantly solves cases when multiple refactorings affect the same program element. Figure 4 presents the composition of two enhanced refactorings, a rename and a move method, that change the same program element, PrintServer.getPacketInfo. Each enhanced refactoring is decomposed into the classic refactoring and its \( \theta \) transformation. The enhanced rename method refactoring changes the arguments of the subsequent move method so that the move method refactoring operates upon element PrintServer.getPacketInfo.

5. Implementation

Programming tools are more likely to be used when they are conveniently incorporated in an Integrated Development Environment (IDE) such as Eclipse. We implemented a semantic, operation-based SCM as an Eclipse plugin, MolhadoRef. MolhadoRef uses the Eclipse Java programming editor as the front end and customizes Molhado framework to store Java programs.

MolhadoRef connects two systems that work in different paradigms. Eclipse editors operate at the file level granularity. Molhado framework allows one to model source code entities at any level of granularity. Also Eclipse offers a name-based refactoring engine whereas MolhadoRef requires an ID-based refactoring engine.
5.1. Molhado Infrastructure

MolhadoRef is built on top of Molhado object-oriented SCM infrastructure [27], which was developed for creating SCM tools. Essentially, Molhado is a database that keeps track of history.

Unlike the file-based SCM approach, Molhado allows an SCM system to model and capture the structure of logical entities within a file and the operations on them. Molhado has a flexible data model that is appropriate for representation of programs in any kind of language; MolhadoRef specializes it for Java.

In Molhado, a program contains a set of nodes, each of which has a set of slots that are attached to it by means of attributes. Nodes are the units of identity, while slots hold values (which can be null) and attributes map nodes to slots. Nodes, slots and attributes that are related to each other form attribute tables. Version control is added into the data model by a third dimension in attribute tables. To represent nodes, Molhado offers two types of components: composite components that can contain other composites and atomic components (the lowest level of granularity).

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 refactoring operations performed by the Eclipse refactoring engine to record the semantics of refactoring operations.

5.2. Eclipse Infrastructure

The Eclipse compare engine offers several APIs for reporting changes at different levels of granularity. MolhadoRef uses Differencer to find changes at the directory or file level. Once it learns the Java files that changed, it uses JavaStructureComparator to report the changes in terms of Java program elements (e.g., classes, methods and fields). From the program elements, the RangeDifferencer finds the low level changes (e.g., changes inside a method body). All the differencers report their results as a tree of DiffNodes, which serve as inputs to JavaStructureDiffViewer that displays graphically the changed elements. Ren et al. [15] present a similar code comparator that moves away from a purely textual representation of changes.

The Eclipse refactoring engine was extended (starting with Eclipse 3.2M4) to record refactoring operations. MolhadoRef uses the new refactoring engine to record and store the performed refactorings. The representation for refactorings in MolhadoRef, which is based on attribute tables as described, uses the XML format of the Eclipse refactoring engine. Therefore, the refactoring operations can be resuscitated and replayed back by the refactoring engine during an update operation.

Eclipse refactorings are based on a processor-participant
architecture. The main bulk of a refactoring analysis is done by the processor, while third parties can contribute new functionality by hooking into well defined extension points and registering their participants. We implemented such participants for all the refactorings mentioned in Section 4.2. 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.

When the user invokes a checkout operation, MolhadoRef reconstructs (from its internal representation) the Java compilation units and packages and invokes the Eclipse code formatter on the files. After MolhadoRef brings the classes and packages into a project in the current Eclipse workspace, the user can resume her programming session using the Eclipse environment.

A detailed description of MolhadoRef’s usage of Molhado infrastructure can be found in a technical report [12].

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 refactoring 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 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 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 changes than 1267 (a reduction of 1 : 19).

Third, being an ID-based SCM, MolhadoRef can always 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 [6, 35]. Early SCM systems (e.g. CVS [26]) provided versioning support for individual files and directories. In addition to version control, advanced SCM systems also provide more powerful configuration management services. Subversion [31] provides more powerful features such as versioning for meta-data, properties of files, renamed or copied files/directories, and cheaper version branching. Similarly, commercial SCM tools still focus on files [35]. 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 [5], POEM [20], Westfechtel’s system [32], Unified Extensional Versioning Model [2], Osth’s fine-grained SCM model [28], etc. However, none of them manage versions of refactored program entities and refactoring operations on those entities in a tightly connected manner as MolhadoRef does.

Software Merging. According to Mens [24], 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 [26], 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, 17, 33]. Other are based on graphs [23, 29]. 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 Wesfetchelt’s context-sensitive merge tool [32], an AST is augmented by the bindings of identifiers to their declarations. More advanced semantic-based merge algorithms [4, 18, 34] detect behavioral conflicts using dependency graphs, program slicing, and denotational semantics.

Operation-based Merging. The operation-based approach has been used in software merging [14, 19, 21, 23, 30]. It is a particular flavor of semantic-based merging that models changes between versions as explicit operations or transformations. Operation-based merge approach can improve conflict detection and allows better conflict solving [24]. Lippe et al. [21] 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 [14]. GINA [3] 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 handle the merging of changes that involve both refactoring and textual editing.

Similar to MolhadoRef, Ekman and Asklund [16] present a refactoring-aware versioning system. Their approach is more lightweight since it keeps the program elements and their IDs in volatile memory, thus allowing for a short-lived history of refactored program entities. Our approach is more heavyweight, 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. Furthermore, their system does not offer support for 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.

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’s refactorings [11, 9]. 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 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 screen shots and download MolhadoRef at: netfiles.uiuc.edu/dig/MolhadoRef

<table>
<thead>
<tr>
<th>CaseStudy</th>
<th>CVS</th>
<th>MolhadoRef</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Conflicts</td>
<td>CompileErrs</td>
</tr>
<tr>
<td>MolhadoRef code</td>
<td>36</td>
<td>41</td>
</tr>
<tr>
<td>MotivatingExample</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. Effectiveness of merging with CVS versus MolhadoRef
A Conflict Detection

We introduce the notations used to describe the conflict scenarios by using the first version of PrintServer program from Section 2 whose AST is shown in Fig. 5. The simpleName denotes the simple name of a node in the AST, while fqName(m) denotes the fully qualified name of a node. The classOf(m) denotes the parent class that defines method mi, defines(mi, mi, mi) is a predicate that becomes true when a class C declares the two methods mi, mi. If the class inherits but does not refine these two methods, the predicate becomes false. ancestorOf(mi) denotes any of the parents of node mi. sig(mi) is the signature of the method mi, defined by the types and the order of the method’s arguments. sig(ce) is the signature of the class ce, defined by the interfaces that the class implements and the superclass that the class inherits from. overridesED(ce, ce) is a predicate that becomes true when the two classes are in an inheritance relationship. overridesN(mi, mi) is a predicate that becomes true when one method overrides another.
or is overridden by the other. The following are the equality conditions for packages (p), classes (c), and methods (m):

\[
p_i = p_j \iff \text{fqName}(p_i) = \text{fqName}(p_j)
\]

\[
c_i = c_j \iff \text{sig}(c_i) = \text{sig}(c_j) \land \text{fqName}(c_i) = \text{fqName}(c_j)
\]

\[
m_i = m_j \iff \text{sig}(m_i) = \text{sig}(m_j) \land \text{fqName}(m_i) = \text{fqName}(m_j)
\]

In general two nodes are equal (identical) when all their properties (including name, fqName, signature, parents) are equal. For the example in Fig. 5:

\[
\text{name} = \text{"accept"}
\]

\[
\text{fqName} = \text{"nodes.PrintServer.accept"}
\]

\[
\text{classOf} = \text{"PrintServer"}
\]

\[
\text{defines} = \text{true}
\]

\[
\text{ancestorOf} = \text{"nodes"}
\]

\[
\text{sig} = \text{sig} = \text{"print"}
\]

\[
\text{inheritsED} = \text{"PrintServer, LANNode"} = \text{true}
\]

\[
\text{overridesN} = \text{true}
\]

The following acronyms are used throughout the sections that follow:

\[
\text{RenP} = \text{Rename Package}
\]

\[
\text{RenC} = \text{Rename Class}
\]

\[
\text{RenM} = \text{Rename Method}
\]

\[
\text{AddP} = \text{Move Package}
\]

\[
\text{AddC} = \text{Move Class}
\]

\[
\text{AddM} = \text{Move Method}
\]

\[
\text{CCS} = \text{Change Class Signature}
\]

\[
\text{CMS} = \text{Change Method Signature}
\]

\[
\text{APD} = \text{Add Package Declaration}
\]

\[
\text{DPD} = \text{Delete Package Declaration}
\]

\[
\text{ACD} = \text{Add Class Declaration}
\]

\[
\text{DCD} = \text{Delete Class Declaration}
\]

\[
\text{AMD} = \text{Add Method Declaration}
\]

\[
\text{DMDe} = \text{Delete Method Declaration}
\]

\[
\text{APC} = \text{Add Package Call (e.g. add import statement)}
\]

\[
\text{DPC} = \text{Delete Package Call (delete import statement)}
\]

\[
\text{ACC} = \text{Add Class Call (e.g. add class instantiation)}
\]

\[
\text{DCC} = \text{Delete Class Call (delete class instantiation)}
\]

\[
\text{AMC} = \text{Add Method Call}
\]

\[
\text{DMC} = \text{Delete Method Call}
\]

A.1 hasConflicts predicates

This section describes the scenarios when two operations, op1 and op2 (denoted op1/op2), result in a conflict, i.e. when the predicate hasConflicts(op1, op2) returns true:

\[
\text{RenM}(m_1, m_2)/\text{RenM}(m_3, m_4):
\]

\[
(m_1 \neq m_3 \land m_2 = m_4) \lor
(m_1 = m_3 \land m_2 \neq m_4) \lor
(\neg \text{overridesN}(m_1, m_3) \land \text{overridesN}(m_2, m_4) \lor
\text{overridesN}(m_1, m_3) \land \neg \text{overridesN}(m_2, m_4))
\]

\[
\text{RenM}(m_1, m_2)/\text{MovM}(m_3, m_4):
\]

\[
(m_1 \neq m_3 \land m_2 = m_4) \lor
(\neg \text{overridesN}(m_1, m_3) \land \text{overridesN}(m_2, m_4) \lor
\text{overridesN}(m_1, m_3) \land \neg \text{overridesN}(m_2, m_4))
\]

\[
\text{RenM}(m_1, m_2)/\text{CMS}(m_3, m_4):
\]

\[
(m_1 \neq m_3 \land m_2 = m_4) \lor
(\neg \text{overridesN}(m_1, m_3) \land \text{overridesN}(m_2, m_4) \lor
\text{overridesN}(m_1, m_3) \land \neg \text{overridesN}(m_2, m_4))
\]

\[
\text{RenM}(m_1, m_2)/\text{AMD}(m_3):
\]

\[
m_2 = m_3 \lor
(\neg \text{overridesN}(m_1, m_3) \land \text{overridesN}(m_2, m_4) \lor
\text{overridesN}(m_1, m_3) \land \neg \text{overridesN}(m_2, m_4))
\]

\[
\text{RenM}(m_1, m_2)/\text{DMDe}(m_3):
\]

\[
m_1 = m_3
\]

\[
\text{RenM}(m_1, m_2)/\text{AMC}(m_3):
\]

false

\[
\text{RenM}(m_1, m_2)/\text{DMC}(m_3):
\]

false

\[
\text{MovM}(m_1, m_2)/\text{DMDe}(m_3):
\]

\[
m_2 = m_3 \lor
(\neg \text{overridesN}(m_1, m_3) \land \text{overridesN}(m_2, m_4) \lor
\text{overridesN}(m_1, m_3) \land \neg \text{overridesN}(m_2, m_4))
\]

\[
\text{MovM}(m_1, m_2)/\text{AMD}(m_3):
\]

\[
m_2 = m_3 \lor
(\neg \text{overridesN}(m_1, m_3) \land \text{overridesN}(m_2, m_4) \lor
\text{overridesN}(m_1, m_3) \land \neg \text{overridesN}(m_2, m_4))
\]

\[
\text{MovM}(m_1, m_2)/\text{AMC}(m_3):
\]

false
\begin{align*}
\text{MovM}(m_1, m_2) &/\text{DMC}(m_3) : \text{false} \\
\text{CMS}(m_1, m_2) &/\text{CMS}(m_3, m_4) : \\
(m_1 = m_3 \land m_2 \neq m_4) \lor \\
(m_1 \neq m_3 \land m_2 = m_4) \lor \\
(\neg \text{overridesN}(m_1, m_3) \land \text{overridesN}(m_2, m_4)) \lor \\
(\text{overridesN}(m_1, m_3) \land \neg \text{overridesN}(m_2, m_4)) \\
\text{CMS}(m_1, m_2) &/\text{AMD}(m_3) : \\
m_2 = m_3 \lor \\
(\neg \text{overridesN}(m_1, m_3) \land \text{overridesN}(m_2, m_3)) \lor \\
(\text{overridesN}(m_1, m_3) \land \neg \text{overridesN}(m_2, m_3)) \\
\text{CMS}(m_1, m_2) &/\text{DMD}(m_3) : \\
m_1 = m_3 \\
\text{CMS}(m_1, m_2) &/\text{AMC}(m_3) : \text{false} \\
\text{CMS}(m_1, m_2) &/\text{DMC}(m_3) : \text{false} \\
\text{AMD}(m_1) &/\text{AMD}(m_2) : \\
m_1 = m_2 \\
\text{AMD}(m_1) &/\text{DMD}(m_2) : \text{false} \\
\text{AMD}(m_1) &/\text{AMC}(m_2) : \text{false} \\
\text{AMD}(m_1) &/\text{DMC}(m_2) : \text{false} \\
\text{DMD}(m_1) &/\text{DMD}(m_2) : \text{false} \\
\text{DMD}(m_1) &/\text{AMC}(m_2) : \\
m_1 = m_2 \\
\text{DMD}(m_1) &/\text{DMC}(m_2) : \text{false} \\
\text{AMC}(m_1) &/\text{AMC}(m_2) : \text{false} \\
\text{AMC}(m_1) &/\text{DMC}(m_2) : \text{false} \\
\text{DMC}(m_1) &/\text{DMC}(m_2) : \text{false} \\
\text{RenC}(c_1, c_2) &/\text{RenC}(c_3, c_4) : \\
(c_1 = c_3 \land c_2 \neq c_4) \lor (c_1 \neq c_3 \land c_2 = c_4) \\
\text{RenC}(c_1, c_2) &/\text{MovC}(c_3, c_4) : \\
(c_1 \neq c_3 \land c_2 = c_4) \\
\text{RenC}(c_1, c_2) &/\text{CCS}(c_3, c_4) : \text{false} \\
\text{RenC}(c_1, c_2) &/\text{ACD}(c_3) : \\
c_2 = c_3 \\
\text{RenC}(c_1, c_2) &/\text{DCD}(c_3) : \text{false} \\
\text{RenC}(c_1, c_2) &/\text{ACC}(c_3) : \text{false} \\
\text{RenC}(c_1, c_2) &/\text{DCC}(c_3) : \text{false} \\
\text{MovC}(c_1, c_2) &/\text{MovC}(c_3, c_4) : \\
(c_1 = c_3 \land c_2 \neq c_4) \lor (c_1 \neq c_3 \land c_2 = c_4) \\
\text{MovC}(c_1, c_2) &/\text{CCS}(c_3, c_4) : \text{false} \\
\text{MovC}(c_1, c_2) &/\text{ACD}(c_3) : \\
c_2 = c_3 \\
\text{MovC}(c_1, c_2) &/\text{DCD}(c_3) : \text{false} \\
\text{MovC}(c_1, c_2) &/\text{ACC}(c_3) : \text{false} \\
\text{MovC}(c_1, c_2) &/\text{DCC}(c_3) : \text{false} \\
\text{CCS}(c_1, c_2) &/\text{CCS}(c_3, c_4) : \\
c_1 = c_3 \land c_2 \neq c_4 \\
\text{CCS}(c_1, c_2) &/\text{ACD}(c_3) : \text{false} \\
\text{CCS}(c_1, c_2) &/\text{DCD}(c_3) : \text{false} \\
\text{CCS}(c_1, c_2) &/\text{ACC}(c_3) : \text{false} \\
\text{CCS}(c_1, c_2) &/\text{DCC}(c_3) : \text{false} \\
\end{align*}
\[ ACD(c_1)/ACD(c_2) : \\
\quad c_1 = c_2 \]

\[ ACD(c_1)/DCD(c_2) : \\
\quad false \]

\[ ACD(c_1)/ACC(c_2) : \\
\quad false \]

\[ ACD(c_1)/DCC(c_2) : \\
\quad false \]

\[ DCD(c_1)/DCD(c_2) : \\
\quad false \]

\[ DCD(c_1)/ACC(c_2) : \\
\quad c_1 = c_2 \]

\[ DCD(c_1)/DCC(c_2) : \\
\quad false \]

\[ ACC(c_1)/ACC(c_2) : \\
\quad false \]

\[ ACC(c_1)/DCC(c_2) : \\
\quad false \]

\[ DCC(c_1)/DCC(c_2) : \\
\quad false \]

\[ \text{RenP}(p_1, p_2)/\text{RenP}(p_3, p_4) : \\
\quad (p_1 = p_3 \land p_2 \neq p_4) \lor (p_1 \neq p_3 \land p_2 = p_4) \]

\[ \text{RenP}(p_1, p_2)/\text{MovP}(p_3, p_4) : \\
\quad (p_1 \neq p_3 \land p_2 = p_4) \]

\[ \text{RenP}(p_1, p_2)/\text{APD}(p_3) : \\
\quad p_2 = p_3 \]

\[ \text{RenP}(p_1, p_2)/\text{DPD}(p_3) : \\
\quad false \]

\[ \text{MovP}(p_1, p_2)/\text{APD}(p_3) : \\
\quad p_2 = p_3 \]

\[ \text{MovP}(p_1, p_2)/\text{DPD}(p_3) : \\
\quad false \]

\[ \text{MovP}(p_1, p_2)/\text{APC}(p_3) : \\
\quad false \]

\[ \text{MovP}(p_1, p_2)/\text{DPC}(p_3) : \\
\quad false \]

\[ \text{APD}(p_1)/\text{APD}(p_2) : \\
\quad p_1 = p_2 \]

\[ \text{APD}(p_1)/\text{DPD}(p_2) : \\
\quad false \]

\[ \text{APD}(p_1)/\text{APC}(p_2) : \\
\quad false \]

\[ \text{APD}(p_1)/\text{DPC}(p_2) : \\
\quad false \]

\[ \text{DPD}(p_1)/\text{DPD}(p_2) : \\
\quad false \]

\[ \text{DPD}(p_1)/\text{APC}(p_2) : \\
\quad p_1 = p_2 \]

\[ \text{DPD}(p_1)/\text{DPC}(p_2) : \\
\quad false \]

\[ \text{APC}(p_1)/\text{APC}(p_2) : \\
\quad false \]

\[ \text{APC}(p_1)/\text{DPC}(p_2) : \\
\quad false \]

\[ \text{DPC}(p_1)/\text{DPC}(p_2) : \\
\quad false \]

\[ \text{A.2. isDependent predicates} \]

This section describes the scenarios when two operations have a dependence. \( op_1 \prec_P op_2 \) indicates that \( op_1 \) must be performed before \( op_2 \), i.e. \( op_2 \) depends on \( op_1 \):

\[ \text{RenP}(p_3, p_4) \prec_P \text{RenP}(p_1, p_2) : \\
\quad \text{ancestorOf}(p_3) = p_1 \]
\[
\begin{align*}
\text{MovP}(p_3, p_4) &\prec_p \text{RenP}(p_1, p_2) : \\
(\text{ancestorOf}(p_3) = p_1) \lor \text{ancestorOf}(p_4) = p_1 \\
\text{RenC}(c_3, c_4) &\prec_p \text{RenP}(p_1, p_2) : \\
\text{ancestorOf}(c_3) = p_1 \\
\text{MovC}(c_3, c_4) &\prec_p \text{RenP}(p_1, p_2) : \\
(\text{ancestorOf}(c_3) = p_1) \lor \text{ancestorOf}(c_4) = p_1 \\
\text{RenM}(m_3, m_4) &\prec_p \text{RenP}(p_1, p_2) : \\
\text{ancestorOf}(m_3) = p_1 \\
\text{CMS}(m_3, m_4) &\prec_p \text{RenP}(p_1, p_2) : \\
\text{ancestorOf}(m_3) = p_1 \\
\text{MovM}(m_3, m_4) &\prec_p \text{RenC}(c_1, c_2) : \\
(\text{classOf}(m_3) = c_1) \lor \text{classOf}(m_4) = c_1 \\
\text{CMS}(m_3, m_4) &\prec_p \text{RenC}(c_1, c_2) : \\
\text{classOf}(m_3) = c_1 \\
\text{AMD}(m_3) &\prec_p \text{RenC}(c_1, c_2) : \\
\text{classOf}(m_3) = c_1 \\
\text{DMD}(m_3) &\prec_p \text{RenC}(c_1, c_2) : \\
\text{classOf}(m_3) = c_1 \\
\text{RenM}(m_3, m_4) &\prec_p \text{MovC}(c_1, c_2) : \\
\text{classOf}(m_3) = c_1 \\
\text{MovM}(m_3, m_4) &\prec_p \text{MovC}(c_1, c_2) : \\
(\text{classOf}(m_3) = c_1) \lor \text{classOf}(m_4) = c_1 \\
\text{CMS}(m_3, m_4) &\prec_p \text{MovC}(c_1, c_2) : \\
\text{classOf}(m_3) = c_1 \\
\text{AMD}(m_3) &\prec_p \text{MovC}(c_1, c_2) : \\
\text{classOf}(m_3) = c_1 \\
\text{DMD}(m_3) &\prec_p \text{MovC}(c_1, c_2) : \\
\text{classOf}(m_3) = c_1 \\
\end{align*}
\]