Refactoring-aware Software Configuration Management

Danny Dig
University of Illinois at Urbana-Champaign
dig@uiuc.edu

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

Ralph Johnson
University of Illinois at Urbana-Champaign
johnson@cs.uiuc.edu

ABSTRACT

Refactoring tools allow programmers to change source code much quicker than before. However, the complexity of these changes cause versioning tools that operate at a file level to lose the history of components. This problem can be solved by semantic, operation-based SCM with persistent IDs. We propose that versioning tools be aware of the program entities and the refactoring operations. MolhadoRef, our prototype, uses these techniques to ensure that it never loses history.

1. INTRODUCTION

Although the first refactoring tool was created over ten years ago [28], they have become popular only in the past few years. Tools like Eclipse [9] and IntelliJ IDEA [15] have made refactoring tools a standard for most Java programmers, and the most recent versions of Microsoft's Visual Studio has started to provide automated support for refactoring for C# programmers. At the beginning of 2006, `refactoring.com` lists refactoring tools for nine different languages.

The wide-spread use of a new kind of software tool often forces other tools to adjust to it. Refactoring tools make particular demands on software configuration management (SCM) tools. A refactoring tool allows a programmer to quickly make changes that potentially affect all parts of a system. Changes that seem simple from a refactoring point of view can be complex from a SCM point of view unless the SCM tools can treat refactorings intelligently.

Refactorings [25] are program transformations that change the structure of a program without changing its external behavior. Programmers refactor a program to simplify it, often by eliminating duplication, or to change its design so that it can more easily accommodate a new feature. Some refactorings are local in scope, such as extracting a function. However, changing the name or interface of a global function can have global scope, since every part of the system that uses the function will have to change. Refactoring tools make it trivial to change the name or interface of a program entity, regardless of how much of the program has to change.

Most SCM systems are based on files, not on logical program structure. They model changes in terms of the lines of a file that have changed, instead of the class or function that changed. In contrast, a semantics-based SCM is tailored to a particular programming language and so can represent individual program entities such as classes and functions. Further, most SCM systems identify components by name, whether it is file name or the name of a program entity. Renaming a component will cause the system to lose track of it. In contrast, Molhado [22] is an SCM infrastructure that creates unique, persistent IDs for each entity, and treats the name as an attribute that can change. Thus, it can track the history of an entity in spite of it being renamed.

An SCM system needs to deal with branches; versions derived from a common base but not from each other. Making branches is easy, but merging them can be hard. File-based SCM systems can merge changes automatically if they are to different parts of a file, but if two branches change the same part of a file then the merge fails and must be done manually. If each branch renames a function and there is a line in the program that calls both functions then a conventional SCM system cannot merge the changes automatically. However, the changes can be merged by a semantics-based SCM that gives methods permanent IDs.

A semantics-based SCM system with persistent IDs is much more suited for refactoring than a traditional SCM system. It helps track the history of refactored, fine-grained program entities, eliminates many conflicts when merging refactored versions in multi-user environments, and represents the history at a higher level. The next section presents an example that illustrates the problems that refactoring causes for conventional SCM systems. Section 3 describes how a semantics-based SCM system with persistent IDs can solve the problem. Refactorings cannot always be merged; section 4 shows when they can. Section 5 describes MolhadoRef, a prototype SCM system that supports refactoring. It is based on Molhado [22], a framework for SCM. The rest of the paper evaluates the prototype and describes what we plan to do next.

2. MOTIVATION EXAMPLE

Consider a simulation of a Local Area Network (LAN) (starting example used in [20]). The system has four versions *v1,v2,v3,* and *v4* as shown in Figure 1. Initially, there are three classes: `Packet`, `LANNode`, and `PrintServer`. 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` refines the behavior of `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. In version *v2*, a developer makes an editing change in the method `PrintServer.getPacketInfo` to produce a new format of the

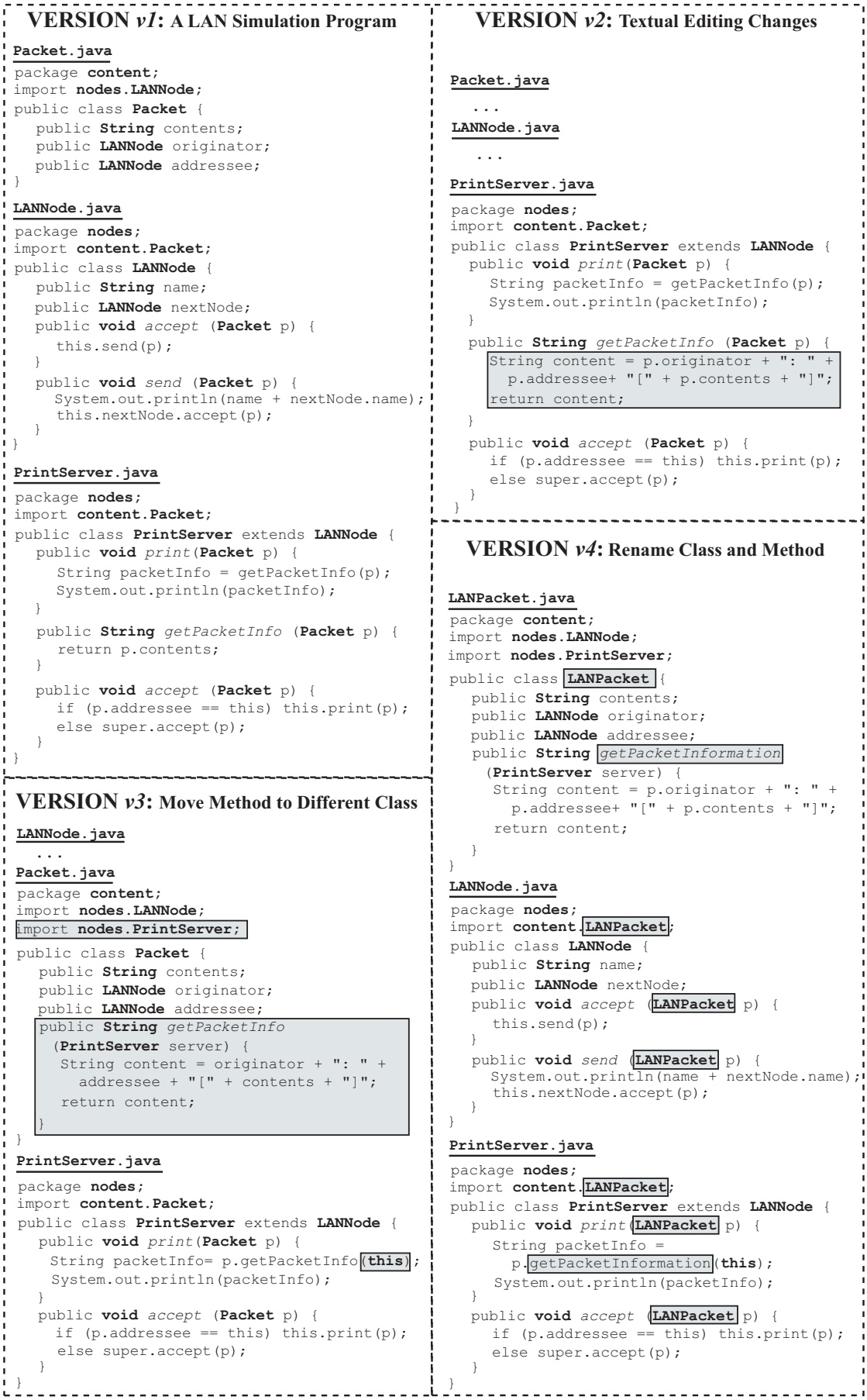


Figure 1: Evolution of Program Entities for a LAN Simulation. Each cell presents the modified code fragments (inside gray boxes) for a particular version.

packet's information. Since method `getPacketInfo` accesses only fields from class `Packet`, in version *v3*, the developer performs a refactoring operation to move method `getPacketInfo` from the class `nodes.PrintServer` to the class `content.Packet`. The import declaration and the signature of the method in the class `Packet` are also updated. The function call in the method `PrintServer.print` of the class is modified by the refactoring engine as well. In version *v4*, one developer renames the method `Packet.getPacketInfo` to `Packet.getPacket-Information` and another, in parallel, renames the class `Packet` into `LANPacket`. These two refactorings require updates to import declarations and function calls in all three classes.

From version *v3* onward, 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 provide much help for a developer to understand code evolution when program entities are refactored. For example, it could not tell that the method `getPacketInformation` of the class `LANPacket` at *v4* has originated from the method `getPacketInfo` of the class `PrintServer`. In addition, since the renaming of the class `Packet` occurs in parallel with the renaming of `getPacketInfo` (by different developers), existing name-based refactoring engines and SCM tools could not create the *merged version* as in version *v4* (see Figure 1).

3. SEMANTICS-BASED, OPERATION-BASED SCM

Traditional SCM systems are *file-based* and designed to support any kind of text files. They treat a software system as a set of files, and a program as a collection of lines of text. They know little about the underlying semantics of file contents. From those SCM repositories, the only thing known about the similarities between versions of a program is the lines that they share. To provide better versioning support for refactored source code, an SCM tool must be able to work at the *semantic level* of programs and refactoring operations. It must be capable of capturing the semantics of program entities and refactoring operations that were performed on those entities. In other words, it must be *semantics-based*.

We developed *MolhadoRef*, a semantics-based SCM system, which is able to capture and version the underlying semantics of Java programs. It also maintains *persistent identifiers* for all program entities in its repository. It uses the *operation-based* SCM approach [17] to represent and record refactoring operations as *first-class* entities in the repository. 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. The operation-based approach gives a precise way to integrate changes caused by editing operations from different lines of parallel development [17, 19]. As for refactoring, recent extensions to refactoring engines [2, 9, 14] allow to record and replay refactoring operations.

MolhadoRef is based on Molhado object-oriented SCM infrastructure and framework [22], which we developed for creating SCM tools. Unlike the file-based approach in traditional SCM systems, Molhado allows a SCM system to model and capture the structure of program entities within a program file and the operations on them. The developers modify their program entities and apply refactoring operations. When they check in the repository, all the changes to program entities as well as refactoring operations applied to them are recorded. The order of refactorings is recorded using timestamps. Developers do not need to worry about the concrete file level since program entities and refactoring operations are directly accessible in the repository. The remainder of this section describes in details the application of Molhado infrastructure to build our SCM system.

3.1 Versioned Data Model

Molhado has a flexible data model that allows it to represent programs in any kind of language. A program consists of 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. This data model is used not only to represent programs and their version history, but also the operations that changed the programs.

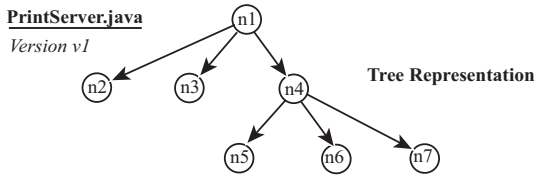
The Molhado data model can be specialized for a particular application. MolhadoRef specializes it to represent Java programs, so nodes are used to represent program entities. The unique identifiers of nodes facilitate the management of entities' histories, especially when they are refactored.

Version control is added into the data model by a third dimension in attribute tables. That is, slots can be versioned. Molhado's version model is called *product versioning* in which a version is *global* across entire system [22]. It allows branching and merging of versions as well. Molhado stores and retrieves versioned attribute tables using techniques derived from the work of Driscoll [8]. No file versioning is involved. More details can be found in [22].

3.2 Capturing Semantics of Program Entities

MolhadoRef captures the semantics of a program with a Molhado component type named *CompilationUnit*, which has a tree-based structure representing the program's Abstract Syntax Tree (AST). The class name is one of its properties. An AST node is represented as a Molhado node. To model the parent node and children nodes of an AST node, each Molhado node is associated with two attributes: "parent" attribute defines for each node the *parent* node in the AST, and "children" attribute defines a sequence of references to its children nodes. In addition to those structural attributes, each Molhado node also has an attribute ("NodeType" attribute) that identifies the syntactical unit represented by that node.

For example, a method is represented by a Molhado node associated with a "NodeType" slot containing "MethodDecl" (see Figure 2). The "NodeType" slots are enumeration values of predefined AST node types. Furthermore, depending on the type of the AST node, the corresponding Molhado node has additional attributes modeling different semantic properties of the AST node. For example, for a "MethodDecl" node, the following attributes are needed: "RetType" (the return type of the method), "MethName" (the method's name), "Modifier" (a string of modifiers of the method such as `public`, `static`, etc), "Parameters" (the method's parameter list), and "SourceCode". Some properties are of the *reference type* containing unique identifiers of components or Molhado nodes. For example, the "SuperType" slot of "n4" contains the unique identifier of the



Attribute Table

	"NodeType"	"Name"	"parent"	"children"
n1	CompiUnit	"PrintServer"	null	[n2,n3,n4]
n2	PackageDecl	"nodes"	n1	null
n3	ImportDecl	"content.Packet"	n1	null
n4	TypeDecl	"PrintServer"	n1	[n5,n6,n7]
n5	MethodDecl	"print"	n4	null
n6	MethodDecl	"getPacketInfo"	n4	null
n7	MethodDecl	"accept"	n4	null

	"SuperType"	"Modifier"	"RetType"	"Parameters"	"SourceCode"
n1	undef	undef	undef	undef	undef
n2	undef	undef	undef	undef	undef
n3	undef	undef	undef	undef	undef
n4	LANNode	"public"	undef	undef	undef
n5	undef	"public"	"void"	[(Packet, p)]	"String packet..."
n6	undef	"public"	"String"	[(Packet, p)]	"return p.cont..."
n7	undef	"public"	"void"	[(Packet, p)]	"if (p.addresse.."

Figure 2: Representation for Class `PrintServer`

LANNode component, which represents the class `LANNode`. If an attribute is not applicable to a node, an *undef* value is used for the corresponding slot. If a node does not have a child, its children slot contains a *null* value. Units that have finer granularity than a method (e.g., statements, expressions) are stored in the method's body, although Molhado enables any level of fine-grained modeling and versioning.

3.3 Representation of Refactoring Operations

MolhadoRef assumes that the front-end editor recognizes refactoring operations that were performed on program entities. When a user commits (checks in) changes to the current version, refactoring operations are recorded along with other changes. Refactoring operations are represented as Molhado components. Parameters of an operation are recorded as attribute values (i.e. slots) associated with the operation. For example, the only refactoring operation performed from *v2* to *v3* in Figure 1 is "MoveMethod". The *MoveMethod* component, has the following properties: 1) "source" referring to the node representing the method that was moved (e.g. "n6" for `getPacketInfo(...)` in Figure 4), 2) "dest" referring to the destination class (e.g. `content.Packet`), 3) "related_variable" referring to the related variable of the move operation (e.g. *p* in `Packet p`), and 4) "target_name" containing the new name of the parameter in the moved method (e.g. the parameter "server" in `getPacketInfo(PrintServer server)` of class `Packet` in *v3*).

Then, "RenameType" operation was performed from *v3* to *v4*. The *RenameType* component, has the following properties: 1) "source" referring to the class node that was renamed (e.g. "m4" in Figure 5), 2) "new_name" containing the new name of the class (e.g. "LANPacket"), and 3) "old_name" containing the old name (e.g "Packet").

PrintServer.java **Version v2**

	"NodeType"	"Name"	...	"SourceCode"
...	...	...	...	...
n6	MethodDecl	"getPacketInfo"	...	"String content = p.origin..."
...	...	...	...	...

Figure 3: Version v2 - Textual Editing Changes

3.4 Fine-grained Versioning Mechanism

Most modern IDEs and editors can be extended to work with MolhadoRef's fine-grained version control by building a "bridge" between the systems, as will be detailed in Section 5. The bridge calls Molhado API functions when the editor changes program entities. Those functions update the values of slots in attribute tables including structural slots such as "children" and"parent".

Consider version *v2* of the LAN Simulation example. The only change between *v1* and *v2* is in the body of the method `getPacketInfo` of class `PrintServer`. As mentioned earlier, program entities that are finer than a method are not modeled as Molhado nodes. When our versioning system is notified about the change of the method body, it updates the "SourceCode" slot associated with node "n6" (see Figure 3).

Figure 4 displays the attribute table for two classes `Packet` and `PrintServer` at *v3* (i.e. after the "MoveMethod" refactoring operation). Changes are highlighted in the picture. The main difference is the relocation of "n6", representing method `getPacketInfo`. Thus, its "parent" slot refers to "m4" and "children" slot of "m4" has an additional child node. Also, the "Parameters" of "n6" now is modified by the refactoring engine (`PrintServer server`). Notice that the "RefactorOps" contains only one refactoring operation, "MoveMethodOp1". Another major change is the addition of the new "ImportDecl" node ("m8") since the refactoring engine inserts an import declaration.

Figure 5 displays the attribute table for two classes `Packet` and `PrintServer` at *v4*. From *v3* to *v4*, there are two refactoring operations that were performed: renaming a type (`Packet`) and renaming a method (`getPacketInfo`). Renaming operations result in the changes to "Name" slots of "m1" (compilation unit node), "m4" (class node), and "n6" (method node). In addition, two refactoring operations "RenType1" and "RenMethod1" are also recorded in the slots corresponding to class and method nodes. Note that the order of two refactoring operations does not matter in this case since the class and the method are identified by "m4" and "n6" respectively. In the attribute table corresponding to `PrintServer`, function calls, import declarations, and method parameters are updated by the refactoring engine, thus, resulting changes to corresponding slots. For example, the "Name" slot of "n3" becomes "content.LANPacket". The attribute table for the class `LANNode` is similar (not shown).

Assume that a user needs to view the version history of the method `getPacketInformation` in the `LANPacket`. Since the method is identified by the node "n6" even when it was moved to a different table (the method was moved to a different class of a different package), its contents at different versions can be easily retrieved. The method is retrieved from the row "n6" of the *LANPacket* table in Figure 5 for version *v4*, and of the *Packet* table (which is the same table) in Figure 4 for version *v3*. At *v3*, the "RefactorOps"

PrintServer.java

	"NodeType"	"Name"	"parent"	"children"	"SuperType"	"Modifier"	"RetType"	"Parameters"	"SourceCode"
n1	CompiUnit	"PrintServer"	null	[n2,n3,n4]	undef	undef	undef	undef	undef
n2	PackageDecl	"nodes"	n1	null	undef	undef	undef	undef	undef
n3	ImportDecl	"content.Packet"	n1	null	undef	undef	undef	undef	undef
n4	TypeDecl	"PrintServer"	n1	[n5,n7]	Node	"public"	undef	undef	undef
n5	MethodDecl	"print"	n4	null	undef	"public"	"void"	[(Packet, p)]	"String packet... p.getPacketInfo (this)...
n7	MethodDecl	"accept"	n4	null	undef	"public"	"void"	[(Packet, p)]	"if (p.addresse.."

Packet.java

	"NodeType"	"Name"	"parent"	"children"	"SuperType"	"Modifier"	"RetType"	"Parameters"	"SourceCode"	"RefactorOps"
m1	CompiUnit	"Packet"	null	[m1,m2, m8,m3]	undef	undef	undef	undef	undef	undef
m2	PackageDecl	"content"	m1	null	undef	undef	undef	undef	undef	undef
m3	ImportDecl	"nodes.LANNode"	m1	null	undef	undef	undef	undef	undef	undef
m4	TypeDecl	"Packet"	m1	[m5,m6, m7,n6]	"Object"	"public"	undef	undef	undef	undef
m5	FieldDecl	"contents"	m4	null	undef	"public"	"String"	undef	undef	undef
m6	FieldDecl	"originator"	m4	null	undef	"public"	LANNode	undef	undef	undef
m7	FieldDecl	"addressee"	m4	null	undef	"public"	LANNode	undef	undef	undef
n6	MethodDecl	"getPacketInfo"	m4	null	undef	"public"	"String"	[(PrintServer, server)]	"String content = originator +..."	**[MoveMethodOp1]**
m8	ImportDecl	"nodes.PrintServer"	m1	null	undef	undef	undef	undef	undef	undef

MoveMethodOp1: {["source": n6], ["dest": m4], ["related_variable": p], ["target_name": "server"]}

Figure 4: Version v3 - Move Method to Different Class

PrintServer.java

	"NodeType"	"Name"	...	"Parameters"	"SourceCode"
n1	CompiUnit	"PrintServer"	...	undef	undef
n2	PackageDecl	"nodes"	...	undef	undef
n3	ImportDecl	'content.LANPac."	...	undef	undef
n4	TypeDecl	"PrintServer"	...	undef	undef
n5	MethodDecl	"print"	...	[(LANPacket, p)]	"String packet... p.getPacketInfor mation(this)...
n7	MethodDecl	"accept"	...	[(LANPacket, p)]	"if (p.addresse.."

LANPacket.java

	"NodeType"	"Name"	...	"SourceCode"	"RefactorOps"
m1	CompiUnit	"LANPacket"	...	undef	undef
m2	PackageDecl	"content"	...	undef	undef
m3	ImportDecl	"nodes.LANNode"	...	undef	undef
m4	TypeDecl	"LANPacket"	...	undef	**[RenType1]**
m5	FieldDecl	"contents"	...	undef	undef
m6	FieldDecl	"originator"	...	undef	undef
m7	FieldDecl	"addressee"	...	undef	undef
n6	MethodDecl	"getPacket- Information"	...	"String content = originator +.."	**[RenMethod1]**
m8	ImportDecl	"nodes.PrintServer"	...	undef	undef

RenType1:{["source": m4], ["new_name": "LANPacket"],["old_name"...]}
RenMethod1:{["source": n6], ["new_name": "getPacketInformation"],[...]}

Figure 5: Version v4 - Renaming Class and Method

slot of "n6" contains a "MoveMethod" operation, and to move backwards in time, a table switch is computed from the "RefactorOps" attribute. Therefore, the method's content at the version *v2* is retrieved from the table *PrintServer* in Figure 3. The method's content at *v1* is retrieved from the table *PrintServer* in Figure 2.

3.5 Discussion

Our approach relies on the existence of logs of refactoring operations. However, logs are not available for the existing versions of software. Also, logs will not be available for all future versions; some developers will not use refactoring engines with recording, and some developers will perform refactorings manually. To exploit the full potential of record & replay of refactorings, we developed RefactoringCrawler [5] to automatically detect the refactorings used to create a new version. These inferred refactorings can be fed into MolhadoRef when recorded refactorings are not available.

The finest level of granularity for program elements in MolhadoRef is method and field declaration. What happens when refactorings like *RenameLocalVariable* and *ExtractMethod* make most of the changes inside the method body? First, renaming a local variable only has effect within the scope of the method body, and is currently treated as an editing operation. Second, refactorings like ExtractMethod that are visible outside of the refactored element are still recorded and stored in MolhadoRef. When they are played back during an update operation, the source code on which they operate is already fully expanded (i.e. it includes the method body as well), thus the refactoring can proceed.

4. COMPOSITION OF REFACTORINGS

Most refactoring engines assume a single user. Refactoring engines make it easy to change a lot of code, but when multiple developers refactor the same code, it is likely that they will create conflicts. The refactorings that one user performed during a programming session might be perfect alone, but are invalid when they are combined with the refactorings that another user performed on the same code. Therefore, a theory for composing refactorings is needed to accommodate multiuser environments.

Currently, no refactoring engine supports dynamic composition of refactorings. We start by presenting the current state of the art research formalism for describing composition of refactorings [27] by extending the definition of refactorings with postconditions. Even though this extension provides a framework to reason about composition of refactorings, it still misses practical cases. We next propose a small extension to the refactoring engines that broadens the possibility of dynamic composition of refactorings. We call this *ID-based refactoring* and it requires little support from the SCM tools.

A refactoring is a program transformation with a precondition that the program must satisfy before the transformation can be applied. For instance, the precondition for deleting a class C is that class C exists and is not referenced.

Several refactorings $R_1, ..., R_n$ can be composed to form another refactoring $R_c = (R_1, ..., R_n)$. When computing the precondition for a chain of refactorings, the naive approach is to AND all the preconditions of individual refactorings:

$pre_{R_c} = pre_{R_1} \wedge ... \wedge pre_{R_n}$

However, this is unnecessarily restrictive since some of the later preconditions might not hold at the start, but only after some of the previous refactorings have been executed. Consider for instance a chain of two refactorings, the first creates class C_1 and the second renames class C_1 to C_2. By simply AND-ing the preconditions, the resulting precondition becomes:

$\neg IsClass(C_1) \wedge isClass(C_1) \wedge \neg IsClass(C_2)$

This formula implies that it is impossible to compose these two refactorings, but it is actually easy.

4.1 Composition with Postconditions

To make refactoring composition possible, Roberts [27] extends the definition of refactoring with *postconditions*:

A refactoring is an ordered triple $R = (pre, T, P)$ where pre is an assertion that must be true on a program for R to be legal, T is the program transformation, and P is a function from assertions to assertions that transforms legal assertions whenever T transforms programs.

The composite refactoring's preconditions are computed by using the information provided by the postcondition assertions. This is done by evaluating the preconditions of each refactoring in the program that has been transformed by earlier refactorings in the composite.

Given a chain of refactorings, $< R_1, R_2, ..., R_n >$, that is legal on program P, the preconditions of R_1 must be true initially, the preconditions of R_2 must be true after the postconditions of R_1 have been applied, and so on. Below are the preconditions for $< AddClass(C1), RenameClass(C1, C2) >$ in an environment where the postcondition (in square brackets) of first refactoring ensures the existence of class C_1:

$\neg IsClass(C_1) \wedge IsClass[C_1/\text{true}](C_1) \wedge \neg IsClass(C_2)$

This formula evaluates to true in an environment where classes C_1 and C_2 do not exist initially.

While taking into account the postconditions can greatly improve the compositionality of refactorings in a single user environment, it does not address the challenges posted by multiuser environments where each user changes the same program without being aware of the other's user changes. In the LAN Simulation example in Section 2, consider a scenario where two users check out the same version. User 1 renames method `LANNode.accept` to `visit` and user 2 renames class `LANNode` to `NetworkNode`. User 1 is the first to check in his code, while user 2 tries to update and check in her changes later. Using the new replay technology, user 2 tries to replay user 1's refactorings on her code. This effectively means composing her rename class refactoring with user 1's rename method. The problem is that although the rename method refactoring was valid in user 1's environment, it is not valid in user 2's environment. This happens because class `LANNode` no longer exists. Mathematically, the composed preconditions of `RenameClass(LANNode, ''NetworkNode'')` and `RenameMethod(LANNode.accept, ''visit'')` are below. Note that postconditions have been added after the execution of `RenameClass`, and for simplicity we only show the first part of the `RenameMethod`'s preconditions:

$IsClass(LANNode) \wedge \neg IsClass(NetworkNode)$
$\wedge IsClass[NetworkNode/true, LANNode/false](LANNode)$
$\wedge ClassDefinesMethod(LANNode, accept)...$

Such preconditions would never hold because class `LANNode` no longer exists in user 2's environment. In the current refactoring engines, the existential assertions are purely based on the names of the source code entities, therefore we call them *name-based refactorings*. Even though in a single user environment name-based refactorings work fine, they fail in environments were multiple users refactor the source code in parallel.

4.2 Composition of ID-based Refactorings

To overcome the shortcomings of *name-based refactoring*, we propose an extension to the refactoring engines called *ID-based refactoring*. We were inspired from the real life of human beings were the citizens in most countries hold a unique ID (e.g., in U.S. this is the Social Security Number) that allows the person's identity to remain the same even though the person might change her name (e.g. through marriage) or relocate to a different part of the country.

We assign to each source code entity a unique ID which remains the same even when the entity is refactored. New IDs get created when new source code entities are added to the program, and IDs get deleted when their corresponding entities get deleted. IDs are stored in the SCM system along with the source code entities when the source code is checked in the repository.

The presence of persistent IDs solves most of the conflicts generated in multi-user environments. Consider the previous example where we compose `RenameClass(LANNode, ''NetworkNode'')` and `RenameMethod(LANNode.accept, ''visit'')`. The ID of class `LANNode` is x1 and the ID of method `LANNode.accept()` is x5. Note that the IDs of these program elements remain the same even after a rename refactoring. The new preconditions with IDs are:

$IsClass(ID = x1, Name = LANNode)$
$\wedge \neg IsClass(ID = undefined, Name = NetworkNode)$
$\wedge IsClass[ID = x1, Name = NetworkNode](ID = x1)$
$\wedge ClassDefinesMethod(ID = x1, ID = x5)...$

Table 1: Conflicts in merging refactoring operations in an ID-based environment when both changes affect the same program entity. Table shows refactoring operations (rename (Ren), move, change method signature (CMS), and editing operations (add/delete method declaration (AMD/DMD), add/delete method call (AMC/DMC). In ID-based environment only conflicts marked with $*$ appear. In addition to these conflicts, new conflicts marked with - appear in a name-based environment.

	Ren	Mov	CMS	AMD	DMD	AMC	DMC
Ren	*	-	-	NA	-	*	-
Mov	*	*	-	NA	-	*	-
CMS	-	-	*	NA	-	*	-
AMD	NA	NA	NA	*	NA	NA	
DMD	*	*	*	NA		*	
AMC				NA	*		
DMC				NA			

These preconditions hold in an environment where class `LANNode` initially exists and there is no class named `NetworkNode`.

IDs can solve several types of conflicts that are unsolvable within the *name-based refactorings* paradigm. We are considering some of the most frequently performed refactorings [6] like renamings (Ren) of methods, classes and packages, moving (Mov) of methods and classes and changing the method signatures (CMS). In addition we are considering some of the most frequent edit operations like adding a method declaration (AMD), deletion of a method declaration (DMD), adding a method call (AMC) and deletion of a method call (DMC). We present the conflict situations in tables that we call *conflict tables*.

In Tables 1, 2, and 3, the vertical left hand-side column presents changes made by user 1. The top row presents changes made by user 2. Both user 1 and user 2 checked out the same version of the source code, and user 1 commits changes first. Before committing, user 2 needs to do an update and merge with user 1's changes. This involves replaying user 1's operations on user 2's code. The $*$ and $-$ symbols represent cases when the operations performed by the two users result in a conflict that cannot be automatically solved, and the system asks the user to resolve the conflict manually. We adopt a conservative approach, that is we do not want to signal a successful merge when the code might not even compile or the semantics of the merged program are changed inadvertently. Due to inheritance and dynamic method dispatching in OO languages, a new range of syntactic or semantic conflicts appear.

All conflicts in an ID-based system will also be conflicts in a name-based system. Conflicts labeled $*$ exist in both kinds of systems, while conflicts labeled $-$ exist only in name-based systems. We present different combinations of operations among source code entities related by three different types of relations: the same entity, inheritance-related and containment-related entities.

Because of space limitations on this paper, we do not describe the conflicts from every combination of operations. A more complete description of all combinations is found in the Appendix.

Table 1 presents the cases when both users performed operations on the same source code entity. Some cases like the one when both users changed the name of the same method (Ren/Ren) require that the last user to commit the changes, picks up the name that is going to be eventually used. Other

Table 2: Conflicts in merging operations that affect inheritance-related source entities (e.g., a method that overrides another method). Conflicts marked with $-$ appear only in a name-based environment.

	Ren	Mov	CMS	AMD	DMD	AMC	DMC
Ren	*	-	-	*	-	*	-
Mov	*	*	*	*	-	*	-
CMS	-	-	*	*	-	*	-
AMD				*	*	*	
DMD		*		*	*	*	
AMC				*	*		
DMC							

cases like the one where user 1 renames a method and user 2 adds a method call (AMC) to the old name, result in a syntactic conflict, but they can be automatically solved by incorporating the AMC operation first and then the rename operation. This way, the rename refactoring operation can find the new call site and correctly update it to the new name. Other cases are simply not applicable (NA). For instance, it cannot happen that the first user that commits the code renames a method which only the second user declared (case Ren/AMD).

All the combinations that are marked with a $-$, can be automatically handled in an environment with ID-based refactorings. For instance, if first user renamed a method and second user changes the method's signature (Ren/CMS), because the changed method is referred to by an unique ID, these two operations can be composed. Note that such a change results in a conflict in a file-based SCM since both users change the same line of code (the method declaration).

Compared with the conflicts in ID-based environment, there are much more conflicts in a name-based environment (marked with both $*$ and $-$). The additional conflicts happen because the source code entity to be changed is identified through its name and signature which might have been changed because of a rename, move or CMS refactoring.

Table 2 presents the cases when the operations are performed on source code entities related by inheritance (e.g., the two users change two methods that override one another). Inheritance and dynamic method dispatch in OO systems create non-trivial challenges. Revisiting the motivating example, consider the scenario where user 1 renames method `LANNode.accept()` to `acceptPacket()` and second user adds a new method declaration `accept()` in the subclass `ColorPrinter` by overriding method `LANNode.accept()`. When composing the two operations, although no syntactic error is signaled, a much subtler semantic error can occur. As a result of the composition, method `ColorPrinter.accept()` no longer overrides `LANNode.acceptPacket()`, thus leading to a different method dispatching than was intended by user 2. The superclass method `LANNode.send(Packet p)` (shown below) fails to dispatch the call to `ColorPrinter.accept()`:

```
protected void send(Packet p) {
   System.out.println(name + nextNode.name);
   this.nextNode.acceptPacket(p);
}
```

Table 3 presents the cases when the two operations affect source code entities related by containment relationship (e.g., one user changes a class and another changes a method within that class). Note that in the ID-based environment there are no conflicts (no $*$ symbols), since any change made

to the name or location of the parent element does not affect changes made in the children elements. However, in the name-based environment (see conflicts marked with −) every change made by user 1 to a parent element is going to effectively conflict with changes made by user 2 to a child element. For instance, in RenP/RenP scenario, user 1 renames package `nodes` to `networkNodes` and user 2 renames subpackage `nodes.printers` to `nodes.networkPrinters`. Composing these refactorings results in a conflict since the parent package `nodes` no longer exists. The Not Applicable (NA) denotes that there cannot be a parent-child relationship among the entities (for instance a method within a method); for space reasons we do not show all the NA cases.

5. TOOL IMPLEMENTATION

Programming tools are more likely to be used in practice when they are conveniently incorporated into an Integrated Development Environment (IDE). IDEs like Eclipse bring all the programming tools to the programmer's fingertips. We implemented a semantic, operation-based SCM as an Eclipse plugin, MolhadoRef. MolhadoRef uses the Eclipse Java programming editor as the front end and the Molhado framework as the SCM back end.

MolhadoRef connects two systems that work in different paradigms. Eclipse editors operate at the file level granularity. Molhado framework models source code entities at a finer level of granularity than file-based systems. Also Eclipse offers a name-based refactoring engine whereas MolhadoRef requires an ID-based refactoring engine.

First, we extended the Molhado framework with support for the program elements found in Java programs as described in Section 3. Molhado offers two types of components: *composite* components that can contain other composites and *atomic* components (the lowest level of granularity). Java packages and compilation units (Java source files) are modeled as Molhado composite components. Program elements within a Java class (e.g., methods, fields) are modeled as atomic components. Although Molhado affords modeling at even finer level of granularity (e.g., program statements and expressions), for efficiency reasons we stop at the method and field declaration level. The name and signature of the method are attributes that allow for distinguishing between possible overloaded methods in the same Java class. Along with methods and fields, inner classes are modeled as entries in the table associated with the main class of a compilation unit; inner classes can expand into new tables when they contain fields and methods.

The interaction with the Eclipse front-end is triggered when a user wants to check in the code. The first time an Eclipse project is checked into in the repository, MolhadoRef does a *lightweight* parsing of the source code. We call it lightweight parsing because it stops at the level of method and field declaration; the parser stores the program statements and expressions within the method bodies or field initializers as a single string. For each Java program entity, MolhadoRef creates its Molhado counterpart. These Molhado entities are connected to form trees mirroring the lightweight Java ASTs. Java source files contain other information like copyright notice and documentation embedded in javadoc comments. Even though technically the documentation is not part of the compiler's AST nodes, we save this information as "documentation" attributes of the corresponding Molhado entities.

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 at the time 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 of the code. This operation-based approach can be very accurate in recording the exact type of change, but a large number of change operations introduce overhead both for recording and replaying. At the other end of the spectrum, in the state-based approach, the deltas are computed 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 gathers 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. The changes caused by refactoring operations are reported by both the compare engine and the refactoring engine. For instance, renaming method `LANPacket.getPacketInfo` to `getPacketInformation`, causes the compare engine to report deletion of `getPacketInfo` and addition of `getPacketInformation`. However, since the refactoring engine also reports the renaming, the change reported by the compare engine is overruled, and the name of the method is updated in tables, thus avoiding loss of history.

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 [26] present a similar code comparator that moves away from a purely textual representation of changes.

The Eclipse refactoring engine was recently extended (starting with Eclipse 3.2M4 of December 2005) 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, is compatible with the XML format that Eclipse refactoring engine uses. Therefore, the refactoring operations can be resuscitated and replayed back by the refactoring engine during an update operation.

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

Table 3: There are no conflicts when merging operations on containment-related source entities (e.g., a package containing another package or class) in an ID-based environment. Capital 'P' reffers to packages, 'C' to classes, 'M' to methods. The conflicts marked with - appear only in a name-based environment.

	RenP	MovP	RenC	MovC	RenM	MovM	AMD	DMD
RenP	-	-	-	-	-	-	-	-
MovP	-	-	-	-	-	-	-	-
RenC			NA	NA	-	-	-	-
MovC			NA	NA	-	-	-	-

Table 4: Evolution of Eclipse's core.refactoring

Version	LOC	Changed LOC	#Pack	#Classes	#Methods
01/31	19933	-	14	114	868
02/28	19993	1786	13	114	871
03/29	20405	526	13	114	875

environment.

The reader can find screen shots and download MolhadoRef at: `netfiles.uiuc.edu/dig/MolhadoRef`

6. EVALUATION

Modeling source code entities and refactoring operations requires extra space when compared with file-based SCM systems. We compare the space required by MolhadoRef with the space used by CVS to keep track of source code.

We checked out of Eclipse CVS repository three revisions of `org.eclipse.ltk.core.refactoring`. This subcomponent is the core of the refactoring engine in Eclipse. These revisions are tagged in the Eclipse repository at 01/31, 02/28/ and 03/29 2006. Table 4 shows how the source code evolved along this time interval. Even though the total number of lines of code does not reveal a great number of changes, the component passed through a great deal of changes revealed by the number of individual lines of code changed. Between versions 01/31 and 02/28, our refactoring-inference engine [5] reveals several structural changes: four classes moved to other packages, one class was renamed, five classes were deleted and five new unrelated classes were added, four methods were renamed and four changed their signatures, one method moved to another class. Between versions 02/28 and 03/29, most changes are edits, e.g., all the classes changed their copyright notice.

Table 5 shows the space taken by the source code relying on the local disk, compared with the space taken by CVS and Molhado. We used the Unix 'disk usage' (du) utility to calculate the total space. We give the size in bytes (as returned by 'du -abc') and in kilobytes ('du -akc'). The Linux machine uses a block size of 1kB for files and 4kB for directories. To calculate the space taken by CVS we checked into our own CVS server the three versions of the source code. As for Molhado, we checked in the three versions. After each check-in we executed a check-out. We used Eclipse comparison utility to verify that our Molhado implementation did not lose/add any source code along the three revisions.

The size in bytes for MolhadoRef degrades gracefully; it is respectively 2.21, 2.69, and 2.80 times larger than the size of initial source code. However, Molhado adds a large number of small files (between 20 and 80 bytes) to represent internally the versioning information. Since the operating system on the machine where we ran the evaluation allocates 1kB for any of these small files, the ratio between the actual files size used to store the source code in Molhado and the initial source code is respectively 3.55, 5.52, and 7.04. On a Windows system that allocated less space for small files, the space on disk for MolhadoRef repository was 2.39, 3.23, and 3.66 times larger than the original source code.

However, given the current trend that disk space is getting cheaper, we believe that the benefits gained by being able to track structural changes far outweigh the extra space. Moreover, it would be possible to implement a more efficient representation for Molhado changes.

MolhadoRef correctly retrieves the history of classes and methods renamed or moved in the 02/28 version, while CVS loses their history. In addition, browsing through the history with MolhadoRef reveals the refactoring operations, thus offering a higher-level understanding of code evolution. CVS shows a lot of changes scattered throughout the source code, with no connection between them.

In the future, when open-source projects will store logs of refactorings performed by different developers, we plan to estimate how many refactorings could be merged within an ID-based environment. We plan to implement a merging algorithm based on the conflict tables (see Section 4.2)and evaluate what is the time saved when merging with our tool. Since there are currently no logs of refactorings for open-source projects, this must wait for the future.

Checking in the source code for the three versions using MolhadoRef took respectively 11, 9 and 7 seconds, while checking out each version took respectively 10, 14, and 14 seconds (check out time includes the compilation of the whole Eclipse project). Using CVS, check-ins took respectively 2, 3, and 2 seconds and each check-out took 8 seconds.

7. RELATED WORK

7.1 SCM Systems

Many sources can be served as excellent surveys on SCM [4, 32]. Early SCM systems (e.g. CVS [21]) provided versioning support for individual files and directories. In addition to version control, advanced SCM systems also provide more powerful configuration management services. Subversion [29] 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 [32].

Several SCM systems have recognized the importance of managing the version history of program entities. Similar to our approach, Gandalf [13] works at the AST level. In Gandalf, each module has a unique interface and potentially multiple realization variants, each of which evolves into versions. In DAMOKLES [7], which is heavily based EER database, leaves of the object composition hierarchy may be as small as statements. POEM [16] provides version control in terms of functions and classes in C++ programs, which are interrelated via a dependency graph that is partitioned into

Table 5: Comparison among spaces required for storing the source code on local disk (no version control), CVS and Molhado. For each system, space size in bytes (B) is the sum of all bytes in every file, while the space used by the operating system to store the files is given in kiloBytes(kB).

Version	Local[B]	Local[kB]	CVS[B]	CVS[kB]	Molhado[B]	Molhado[kB]
01/31	722596	984	724717	968	1602252	3500
02/28	718026	968	832425	1104	1934408	5344
03/29	735525	988	869721	1136	2063352	6964

work areas. The unique identifiers in MolhadoRef is similar in spirit to unique tags for AST nodes in Westfechtel's system [31]. In that system, tags are used in incremental updating of revisions of dependent documents.

The tree-based versioning framework in COOP/Orm [18] works not only for programs but also for hierarchically structured documents. The principles of the framework include sharing unchanged nodes among versions and change propagation. The Unified Extensional Versioning Model [1] supports fine-grained versioning for a tree-structured document by composite, atomic, and link nodes. Each atomic node is represented via a textual file. Since their focus is on collaborative and interactive editing tasks, fine-grained changes are not persistent. In Coven [3], the exact size of a versioned fragment depends on the supported document: entire method and field declaration for C++ and Java programs, or paragraph of text in LaTeX documents. A project in Coven is a composition of *compound artifacts*, which are sets of other artifacts and/or program fragments. In Ohst's fine-grained SCM model [23], changes are managed within contexts of UML *tools* and *design transactions*.

The operation-based approach has been used in software merging [17]. It is a particular flavor of change-based merging that models changes as explicit operations or transformations. Operation-based merge approach can improve conflict detection [19]. Lippe *et al* [17] described a theoretical framework for conflict detection with respect to general transformations. No concrete application or tool for refactorings was presented. Edwards' operation-based framework [10] detects and resolves semantic conflicts from application supplied semantics of operations. However, no existing SCM tool is able to manage versions of fine-grained program entities and refactoring operations performed on those entities in a tightly connected manner as in MolhadoRef.

7.2 Refactoring

Programmers have been cleaning up their code for decades, though the term *refactoring* was coined much later [25]. Opdyke [24] wrote the first catalog of refactorings while Roberts and Brant [28] were the first to implement a refactoring engine. The refactoring field gained much popularity with the catalog of refactorings written by Fowler et al. [12]. Soon after this, IDEs began to incorporate refactoring engines. Tokuda and Batory [30] describe the large architectural changes in two frameworks as a large sequence of small refactorings. They estimate that automated refactorings are 10 times quicker to perform than manual ones.

Record-and-replay of refactorings was recently demonstrated in CatchUp [14] and JBuilder2005 [2] and is planned to be a standard part of Eclipse 3.2. Our methodology uses the record-and-replay of refactorings, although there are many more components needed to build a refactoring-aware SCM.

Ekman and Asklund [11] insightfully present the benefits of refactoring-aware versioning systems: the ability to track the history of refactored program entities, better merging in the presence of well defined semantics of refactoring operations, and better human understanding of the code evolution. They too present a programming model that affords refactoring-aware versioning system for Eclipse. Our approaches are different in many ways: their model heavily relies on modifications to the Eclipse *front-end* (e.g., changing the Eclipse Java Model class hierarchy to support IDs for program elements), whereas we rely on a powerful *back-end* SCM to model program entities with unique IDs. Since we do not impose any changes to the development environment, our approach can be smoothly integrated with other IDEs (in fact we had another implementation with a standalone front-end editor). 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 lifecycle of the software project allowing for a global history tracking of refactored entities.

8. CONCLUSION & FUTURE WORK

Refactoring tools have become popular because they allow programmers to safely make structural changes in large systems. However, such changes create problems for the current SCM tools that operate at the file level. As a result, the history of refactored entities is lost. We propose a novel SCM system, MolhadoRef, that is aware of program entities and the refactoring operations that change them. Because MolhadoRef uses a unique identifier for each program element, it can track the history of refactored program elements. In addition, we introduce the notion of *ID-based refactoring* and we show how unique identifiers allow for many more merging scenarios in multiuser environments than traditional name-based refactoring.

We implemented a refactoring-aware SCM as MolhadoRef, an Eclipse plugin. We extended the Molhado framework to model Java program elements and store refactoring information. By evaluating the extra space required to model program elements and refactoring operations, we learned that the extra space is around three times larger than the original source code. We believe the benefits of tracking refactored entities far outweigh the extra space cost.

Our approach requires a mapping mechanism between the refactoring-aware, ID-based SCM tool and a name-based editing environment. Constructing such a mechanism is not trivial, as described in Section 5.

In the future, we plan to focus on algorithms for semantic merging of refactoring operations and regular edit operations, so that the level of user involvement is minimal. Because we operate at the semantical level of changes, the merging is going to be much more powerful than traditional line-based merging. Also, we are looking into further reducing the extra disk space required by our semantic SCM.

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

9. ACKNOWLEDGMENTS

We would like to thank Darko Marinov, Adam Kiezun, Russ Ruffer, Joe Yoder, Paul Adamczyk and members of Software Architecture Group at UIUC for reviewing drafts of this paper. This work is partially funded through an Eclipse Innovation Grant for which we are very grateful to IBM corporation.

10. REFERENCES

[1] U. Asklund, L. Bendix, H. Christensen, and B. Magnusson. The unified extensional versioning model. In *Proceedings of the 9th Software Configuration Management Workshop*. Springer Verlag, 1999.

[2] www.borland.com/resources/en/pdf/white_papers/jb2005_whats_new.pdf.

[3] M. C. Chu-Carroll, J. Wright, and D. Shields. Supporting aggregation in fine grained software configuration management. In *Proceedings of the tenth Foundations of software engineering symposium*, pages 99–108. ACM Press, 2002.

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

[5] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. In *to appear ECOOP'06: European Conference on OO Programming*, 2006.

[6] D. Dig and R. Johnson. The role of refactorings in api evolution. In *ICSM'05: Proceedings of International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[7] K. Dittrich, W. Gotthard, and P. Lockemann. DAMOKLES: a database sytem for software engineering environments. In *Proceedings of the International Workshop on Advanced Programming Environments*. Springer Verlag, 1986.

[8] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, Feb. 1989.

[9] Eclipse Foundation. http://eclipse.org.

[10] W. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. In *Proceedings of Symposium User Interface Software Technology*, 1997.

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

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Adison-Wesley, 1999.

[13] N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, Dec 1986.

[14] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE'05: Proceedings of International Conference on Software Engineering*, pages 274–283, 2005.

[15] JetBrains Corp. http://www.jetbrains.com/idea.

[16] Y. Lin and S. Reiss. Configuration management with logical structures. In *ICSE'96: Proceedings of International Conference on Software Engineering*, pages 298–307, 1996.

[17] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE5: Proceedings of Symposium on Software Development Environments*, pages 78–87. ACM Press, 1992.

[18] B. Magnusson and U. Asklund. Fine-grained revision control of Configurations in COOP/Orm. In *Proceedings of the 6th Software Configuration Management Workshop*. Springer Verlag, 1996.

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

[20] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, July 2005.

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

[22] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *ICSE'05: Proceedings of International Conference on Software Engineering*, pages 215–224. ACM Press, 2005.

[23] D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. In *FSE'03: Proceedings of the Foundations of software engineering*, pages 227–236. ACM Press, 2003.

[24] B. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, U of Illinois at Urbana-Champaign, 1992.

[25] B. Opdyke and R. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA'90: Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[26] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA'04*, pages 432–448, 2004.

[27] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, Univ. of Illinois at Urbana-Champaign, 1999.

[28] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for smalltalk. *TAPOS*, 3(4):253–263, 1997.

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

[30] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, January 2001.

[31] B. Westfechtel. Revision Control in an Integrated Software Development Environment. In *Proceedings of the 2nd Software Configuration Management Workshop*, pages 96–105. ACM Press, 1989.

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

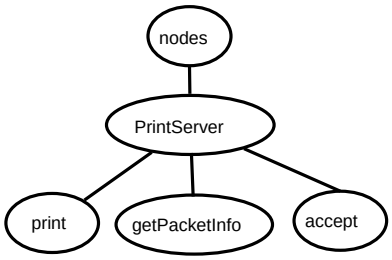


Figure 6: AST representation

APPENDIX

A. CONFLICT TABLES

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. 6. *name* denotes the simple name of a node in the AST, while $fqName(m)$ denotes the fully qualified name of a node. $classOf(m_i)$ denotes the parent class that defines method m_i. Class c_i defines(m_i, m_j) is a predicate that becomes true when a class declares the two methods. If the class inherits but does not refine these two methods, the predicate becomes false. $parentOf(m_i)$ denotes any of the parents of node m_i. $sig(m_i)$ is the signature of the method m_i, defined by the types and the order of the method's arguments. $c_i isSub/SuperClassOf(c_j)$ is a predicate that becomes true when the two classes are in an inheritance relationship. $m_i override/n(m_j)$ (to be read override or overridden) is a predicate that becomes true when one method overrides or is overridden by the other. Two nodes are equal (identical) when all their properties (including name, fqName, signature, parents) are equal. For the example in Fig. 6:

$name(accept) =$ "$accept$"
$fqName(accept) =$ "$nodes.PrintServer.accept$"
$classOf(accept) = PrintServer$
$PrintServer defines(accept, print) = true$
$parentOf(accept) = PrintServer$
$parentOf(accept) = nodes$
$parentOf(PrintServer) = nodes$
$sig(accept) = sig(print)$
$PrintServer isSub/SuperClassOf(LANNode) = true$
$PrintServer.accept override/n(LANNode.accept)$

A.1 Same Entity

This expands the conflicts in Table 1:

$Ren(m_1, m_2)/Ren(m_3, m_4)$:
$((m_1 = m_3) \wedge (name(m_2) \neq name(m_4))) \bigvee$
$((m_1 \neq m_3) \wedge \exists Class(C).Cdefines(m_1, m_3)$
$\wedge sig(m_1) = sig(m_3) \wedge (name(m_2) = name(m_4)))$

$Ren(m_1, m_2)/Mov(m_3, m_4)$:
$((m_1 = m_3) \bigvee$
$((m_1 \neq m_3) \wedge sig(m_2) = sig(m_4)$
$\wedge (name(m_2) = name(m_4)) \wedge (Class(C) defines(m_2, m_4))$

$Ren(m_1, m_2)/CMS(m_3, m_4)$:
$((m_1 = m_3) \bigvee$
$((m_1 \neq m_3) \wedge \exists Class(C).Cdefines(m_1, m_3)$
$\wedge sig(m_2) = sig(m_4) \wedge (name(m_2) = name(m_4)))$

$Ren(m_1, m_2)/DMD(m_3)$:
$m_1 = m_3$

$Ren(m_1, m_2)/AMC(m_3)$:
$m_1 = m_3$

$Ren(m_1, m_2)/DMC(m_3)$:
$m_1 = m_3$

$Mov(m_1, m_2)/Ren(m_3, m_4)$:
$(m_1 = m_3) \bigvee$
$((m_1 \neq m_3) \wedge (classOf(m_2) = classOf(m_4))$
$\wedge (name(m_2) = name(m_4)) \wedge (sig(m_2) = sig(m_4)))$

$Mov(m_1, m_2)/Mov(m_3, m_4)$:
$((m_1 = m_3) \wedge (classOf(m_2) \neq classOf(m_4))) \bigvee$
$((m_1 \neq m_3) \wedge (classOf(m_2) = classOf(m_4))$
$\wedge (name(m_2) = name(m_4)) \wedge (sig(m_2) = sig(m_4)))$

$Mov(m_1, m_2)/CMS(m_3, m_4)$:
$(m_1 = m_3) \bigvee$
$((m_1 \neq m_3) \wedge (classOf(m_2) = classOf(m_4))$
$\wedge (name(m_2) = name(m_4)) \wedge (sig(m_2) = sig(m_4)))$

$Mov(m_1, m_2)/DMD(m_3)$:
$(m_1 = m_3)$

$Mov(m_1, m_2)/AMC(m_3)$:
$m_1 = m_3$

$Mov(m_1, m_2)/DMC(m_3)$:
$m_1 = m_3$

$CMS(m_1, m_2)/Ren(m_3, m_4)$:
$((m_1 = m_3) \bigvee$
$((m_1 \neq m_3) \wedge \exists Class(C).Cdefines(m_1, m_3)$
$\wedge sig(m_2) = sig(m_4) \wedge (name(m_2) = name(m_4)))$

$CMS(m_1, m_2)/Mov(m_3, m_4)$:
$(m_1 = m_3) \bigvee$
$((m_1 \neq m_3) \wedge \ (classOf(m_2) = classOf(m_4))$
$\wedge sig(m_2) = sig(m_4) \wedge (name(m_2) = name(m_4)))$

$CMS(m_1, m_2)/CMS(m_3, m_4)$:
$((m_1 = m_3) \wedge (sig(m_2) \neq sig(m_4))) \bigvee$
$((m_1 \neq m_3) \wedge \ (classOf(m_1) = classOf(m_3))$
$\wedge sig(m_2) = sig(m_4) \wedge (name(m_2) = name(m_4)))$

$CMS(m_1, m_2)/DMD(m_3)$:
$m_1 = m_3$

$CMS(m_1, m_2)/DMC(m_3)$:
$m_1 = m_3$

$CMS(m_1, m_2)/AMC(m_3)$:
$m_1 = m_3$

$DMD(m_1)/Ren(m_3, m_4)$:
$m_1 = m_3 \bigvee$
$m_1 = m_4$

$DMD(m_1)/Mov(m_3, m_4)$:
$m_1 = m_3 \bigvee$
$((m_1 \neq m_3) \wedge (classOf(m_1) = classOf(m_4))$
$\wedge (name(m_1) = name(m_4)) \wedge (sig(m_1) = sig(m_4)))$

$DMD(m_1)/CMS(m_3, m_4)$:
$m_1 = m_3 \bigvee$
$((m_1 \neq m_3) \wedge (classOf(m_1) = classOf(m_3))$
$\wedge (name(m_1) = name(m_4)) \wedge (sig(m_1) = sig(m_4)))$

$DMD(m_1)/AMC(m_3)$:
$m_1 = m_3$

$AMC(m_1)/DMD(m_3)$:
$m_1 = m_3$

A.2 Inheritance-Related

This expands the cells in Table 2:

$Ren(m_1, m_2)/Ren(m_3, m_4)$:
$((m_1 override/n(m_3)) \wedge (name(m_3) \neq name(m_4)) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_4)$
$\wedge sig(m_2) = sig(m_4)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$Ren(m_1, m_2)/Mov(m_3, m_4)$:
$(m_1 override/n(m_3)) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_4)$
$\wedge sig(m_2) = sig(m_4)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$Ren(m_1, m_2)/CMS(m_3, m_4)$:
$(m_1 override/n(m_3)) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_4)$
$\wedge sig(m_2) = sig(m_4)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$Ren(m_1, m_2)/AMD(m_3)$:
$(m_1 override/n(m_3)) \wedge (name(m_2) \neq name(m_3)) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_3)$
$\wedge sig(m_2) = sig(m_3)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$Ren(m_1, m_2)/DMD(m_3)$:
$m_1 override/n(m_3)$

$Ren(m_1, m_2)/AMC(m_3)$:
$m_1 override/n(m_3)$

$Ren(m_1, m_2)/DMC(m_3)$:
$m_1 override/n(m_3)$

$Mov(m_1, m_2)/Ren(m_3, m_4)$:
$(m_1 override/n(m_3)) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_4)$
$\wedge sig(m_2) = sig(m_4)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$Mov(m_1, m_2)/Mov(m_3, m_4)$:
$((m_1 override/n(m_3))$
$\wedge classOf(m_2) isSub/SuperClass(classOf(m_4))) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_4)$
$\wedge sig(m_2) = sig(m_4)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$Mov(m_1, m_2)/CMS(m_3, m_4)$:
$(m_1 override/n(m_3)) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_4)$
$\wedge sig(m_2) = sig(m_4)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$Mov(m_1, m_2)/AMD(m_3)$:
$((m_1 override/n(m_3)) \wedge (m_2 override/n(m_3))) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_4)$
$\wedge sig(m_2) = sig(m_4)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$Mov(m_1, m_2)/DMD(m_3)$:
$m_1 override/n(m_3)$

$Mov(m_1, m_2)/AMC(m_3)$:
$m_1 override/n(m_3)$

$Mov(m_1, m_2)/DMC(m_3)$:
$m_1 override/n(m_3)$

$CMS(m_1, m_2)/Ren(m_3, m_4)$:
$(m_1 override/n(m_3)) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_4)$
$\wedge sig(m_2) = sig(m_4)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$CMS(m_1, m_2)/Mov(m_3, m_4)$:
$(m_1 override/n(m_3)) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_4)$
$\wedge sig(m_2) = sig(m_4)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$CMS(m_1, m_2)/CMS(m_3, m_4)$:
$((m_1 override/n(m_3)) \wedge (sig(m_2) \neq sig(m_4))) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_4)$
$\wedge sig(m_2) = sig(m_4)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$CMS(m_1, m_2)/AMD(m_3)$:
$(m_1 override/n(m_3)) \wedge (sig(m_2) \neq sig(m_3)) \bigvee$
$((m_1 \neg override/n(m_3)) \wedge name(m_2) = name(m_3)$
$\wedge sig(m_2) = sig(m_3)$
$\wedge classOf(m_2) isSub/SuperClassOf(classOf(m_4)))$

$CMS(m_1, m_2)/DMD(m_3)$:
$m_1 override/n(m_3)$

$CMS(m_1, m_2)/AMC(m_3)$:
$m_1 override/n(m_3)$

$CMS(m_1, m_2)/DMC(m_3)$:
$m_1 override/n(m_3)$

The remaining cells in this subsection cannot be determined if they result in conflict at compile time, but only at runtime:

$AMD(m_1)/AMD(m_3)$:
$name(m_1) = name(m_3)$

$\wedge\, sig(m_1) = sig(m_3)$
$\wedge\, classOf(m_1) isSub/SuperClassOf(classOf(m_3))$

$AMD(m_1)/DMD(m_3)$:
$name(m_1) = name(m_3)$
$\wedge\, sig(m_1) = sig(m_3)$
$\wedge\, classOf(m_1) isSub/SuperClassOf(classOf(m_3))$

$AMD(m_1)/AMC(m_3)$:
$name(m_1) = name(m_3)$
$\wedge\, sig(m_1) = sig(m_3)$
$\wedge\, classOf(m_1) isSub/SuperClassOf(classOf(m_3))$

$DMD(m_1)/Mov(m_3, m_4)$:
$name(m_1) = name(m_3)$
$\wedge\, sig(m_1) = sig(m_3)$
$\wedge\, classOf(m_1) isSub/SuperClassOf(classOf(m_3))$

$DMD(m_1)/AMD(m_3)$:
$name(m_1) = name(m_3)$
$\wedge\, sig(m_1) = sig(m_3)$
$\wedge\, classOf(m_1) isSub/SuperClassOf(classOf(m_3))$

$DMD(m_1)/DMD(m_3)$:
$name(m_1) = name(m_3)$
$\wedge\, sig(m_1) = sig(m_3)$
$\wedge\, classOf(m_1) isSub/SuperClassOf(classOf(m_3))$

$DMD(m_1)/AMC(m_3)$:
$name(m_1) = name(m_3)$
$\wedge\, sig(m_1) = sig(m_3)$
$\wedge\, classOf(m_1) isSub/SuperClassOf(classOf(m_3))$

$AMC(m_1)/AMD(m_3)$:
$name(m_1) = name(m_3)$
$\wedge\, sig(m_1) = sig(m_3)$
$\wedge\, classOf(m_1) isSub/SuperClassOf(classOf(m_3))$

$AMC(m_1)/DMD(m_3)$:
$name(m_1) = name(m_3)$
$\wedge\, sig(m_1) = sig(m_3)$
$\wedge\, classOf(m_1) isSub/SuperClassOf(classOf(m_3))$

A.3 Containment-Related

This expands the conflict cells in Table 3:

$RenP(p_1, p_2)/RenP(p_3, p_4)$:
$parentOf(p_3) = p_1$

$RenP(p_1, p_2)/MovP(p_3, p_4)$:
$(parentOf(p_3) = p_1) \bigvee$
$(parentOf(p_3) \neq p_1 \wedge parentOf(p_4) = p_2$
$\wedge\, \exists p_i.parentOf(p_i) = p_2 \wedge fqName(p_i) = fqName(p_4))$

$RenP(p_1, p_2)/RenC(c_3, c_4)$:
$parentOf(c_3) = p_1$

$RenP(p_1, p_2)/MovC(c_3, c_4)$:
$(parentOf(c_3) = p_1) \bigvee$
$(parentOf(c_3) \neq p_1 \wedge parentOf(c_4) = p_2$
$\wedge\, \exists c_i.parentOf(c_i) = p_2 \wedge fqName(c_i) = fqName(c_4))$

$RenP(p_1, p_2)/RenM(m_3, m_4)$:
$parentOf(m_3) = p_1$

$RenP(p_1, p_2)/CMS(m_3, m_4)$:
$parentOf(m_3) = p_1$

$RenP(p_1, p_2)/Mov(m_3, m_4)$:
$(parentOf(m_3) = p_1) \bigvee$
$(parentOf(m_3) \neq p_1 \wedge parentOf(m_4) = p_2$
$\wedge\, \exists c_i.parentOf(c_i) = p_2 \wedge name(m_i) = name(m_4)$
$\wedge\, sig(m_1) = sig(m_4) \wedge c_i defines(m_i, m_4))$

$RenP(p_1, p_2)/AMD(m_3)$:
$parentOf(m_3) = p_1$

$RenP(p_1, p_2)/DMD(m_3)$:
$parentOf(m_3) = p_1$

$MovP(p_1, p_2)/RenP(p_3, p_4)$:
$parentOf(p_3) = p_1$

$MovP(p_1, p_2)/MovP(p_3, p_4)$:
$parentOf(p_3) = p_1$

$MovP(p_1, p_2)/RenC(c_3, c_4)$:
$parentOf(c_3) = p_1$

$MovP(p_1, p_2)/MovC(c_3, c_4)$:
$(parentOf(c_3) = p_1) \bigvee$
$(parentOf(c_3) \neq p_1 \wedge parentOf(c_4) = p_2$
$\wedge\, \exists c_i.parentOf(c_i) = p_2 \wedge fqName(c_i) = fqName(c_4))$

$MovP(p_1, p_2)/RenM(m_3, m_4)$:
$parentOf(m_3) = p_1$

$MovP(p_1, p_2)/CMS(m_3, m_4)$:
$parentOf(m_3) = p_1$

$MovP(p_1, p_2)/MovM(m_3, m_4)$:
$(parentOf(m_3) = p_1) \bigvee$
$(parentOf(m_3) \neq p_1 \wedge parentOf(m_4) = p_2$
$\wedge\, \exists c_i.parentOf(c_i) = p_2 \wedge name(m_i) = name(m_4)$
$\wedge\, sig(m_1) = sig(m_4) \wedge c_i defines(m_i, m_4))$

$MovP(p_1, p_2)/AMD(m_3)$:
$parentOf(m_3) = p_1$

$MovP(p_1, p_2)/DMD(m_3)$:
$parentOf(m_3) = p_1$

$RenC(c_1, c_2)/RenM(m_3, m_4)$:
$classOf(m_3) = c_1$

$RenC(c_1, c_2)/MovM(m_3, m_4)$:
$(classOf(m_3) = c_1) \bigvee$
$(classOf(m_3) \neq c_1 \wedge classOf(m_4) = c_2$
$\wedge\, \exists m_i.classOf(m_i) = c_2$
$\wedge\, name(m_i) = name(m_4) \wedge sig(m_i) = sig(m_4))$

$RenC(c_1, c_2)/CMS(m_3, m_4)$:
$classOf(m_3) = c_1$

$RenC(c_1, c_2)/AMD(m_3)$:

$classOf(m_3) = c_1$

$RenC(c_1, c_2)/DMD(m_3)$:
$classOf(m_3) = c_1$

$MovC(c_1, c_2)/RenM(m_3, m_4)$:
$classOf(m_3) = c_1$

$MovC(c_1, c_2)/MovM(m_3, m_4)$:
$(classOf(m_3) = c_1) \bigvee$
$(classOf(m_3) \neq c_1 \wedge classOf(m_4) = c_2$
$\wedge \exists m_i.classOf(m_i) = c_2$
$\wedge name(m_i) = name(m_4) \wedge sig(m_i) = sig(m_4))$

$MovC(c_1, c_2)/CMS(m_3, m_4)$:
$classOf(m_3) = c_1$

$MovC(c_1, c_2)/AMD(m_3)$:
$classOf(m_3) = c_1$

$MovC(c_1, c_2)/DMD(m_3)$:
$classOf(m_3) = c_1$