Differential Refactoring Engines

Jeffrey L. Overbey and Ralph E. Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave. MC 258
Urbana, IL 61801
{overbey2,johnson}@cs.uiuc.edu

ABSTRACT

The complexities of production programming languages make refactoring tools difficult to develop and notoriously bug-ridden in practice. This paper proposes a new design for refactoring tools in which the refactored program is analyzed *after* it has been transformed rather than before. Instead of checking an *ad hoc* set of preconditions, refactorings simply transform the program; then, a generic analysis procedure attempts to determine whether behavior was preserved. This makes the specification and implementation of automated refactorings both simpler and more robust. We used this technique to construct 18 refactorings across three target languages. Compared with traditional specifications of these refactorings, the total number of precondition checking steps decreased by nearly 90%, and the types of errors automatically detected include some highly nontrivial bugs previously identified in the literature.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.4 [**Programming Languages**]: Processors

General Terms

Languages

Keywords

Program representation, refactoring

1. INTRODUCTION

The most common refactorings are simple program transformations, at least conceptually. Rename, Extract, Move, Change Signature, Inline, Encapsulate, Pull Up, Push Down: all of these are, in essence, sophisticated variations on find-and-replace or cut-and-paste. This apparent simplicity is deceptive, however. Consider the Eclipse JDT—an extremely popular, very mature tool frequently cited in the refactoring literature (e.g., [1, 7, 8, 11, 12]). It implements all of these refactorings. How many have known errors in their implementations? All of them.

Automated refactoring tools operate on source code, and modern programming languages are complex, so some complexity in a refactoring tool is inevitable. However, the plethora of bugs in current tools is not a necessary consequence. This paper suggests that *much of the complexity and error in automated refactoring tools can be mitigated by fundamentally changing their design.*

1.1 The Problem with Preconditions

An automated refactoring has two parts. One part is the transformation—the change it makes to the source code. The other is a set of preconditions which are checked before the transformation is applied. The preconditions guarantee that the transformation will produce a program that compiles and executes with the same behavior as the original program.

Designing a sufficient set of preconditions is extremely difficult. The author of the refactoring must exhaustively consider every language feature, every syntactic construct, every extragrammatical restriction, every semantic rule... and somehow guarantee that the transformation is incapable of producing an error. Although such assurances are possible in theory, the size and complexity of modern programming languages makes them tremendously difficult to make in practice. Consider Java: Even a "simple" refactoring like Rename must consider naming conflicts, namespaces, qualifiers, shadowing, reserved words, inheritance, overriding, overloading, constructors, visibility, inner classes, reflection, externally-visible names, and "special" names such as `main`.

1.2 The Differential Solution

In this paper, we will observe that preconditions guarantee three basic properties—*input validity, compilability,* and *behavior preservation*—and use this observation to propose a new design for refactoring engines. In this design, compilability and preservation preconditions are not checked explicitly. Instead, the transformation is performed, and afterward the refactoring engine attempts to determine whether compilability and preservation were maintained by contrasting a *semantic model* of the transformed program with a similar model of the original program. We call this design a *differential refactoring engine.* It has two major advantages over traditional designs:

- *A surprisingly large number of precondition checks can be eliminated.* Often, the refactoring does not need

ICSE '11 Honolulu, Hawaii USA

to explicitly check any compilability or preservation preconditions at all. This greatly simplifies the specification and implementation of these refactorings.

- *Source code is analyzed after transformation, as well as before.* This makes it possible for the tool to detect certain errors in the transformed code, and it provides a sanity check for the transformation, notifying the user if a buggy transformation will make erroneous changes to their code.

The primary contribution of this paper is the idea that refactorings can be implemented by checking for compilability and preservation *after* transformation in a *generic* way—i.e., compilability and preservation can be checked in the same way for every refactoring. This, in turn, simplifies individual refactorings by eliminating the need to explicitly design preconditions to check for compilability and preservation. Following a discussion of precondition checking in §2, the general structure of a differential refactoring engine is given in §3. To illustrate the feasibility of the differential approach, we propose one possible design in §5 and demonstrate that it is sufficient to describe common refactorings in §6.

2. PRECONDITIONS AND REFACTORING

2.1 Validity, Compilability, and Preservation

A refactoring's preconditions determine conditions under which the program transformation will preserve behavior. This means that preconditions guarantee three properties:

1. *Input validity.* All input from the user is legal; it is possible to apply the transformation to the given program with the given inputs.
2. *Compilability.* If the transformation is performed, the resulting program will compile; it will meet all the syntactic and semantic requirements of the target language.
3. *Preservation.* If the transformation is performed and the resulting program is compiled and executed, it will exhibit the same runtime behavior as the untransformed program.

2.2 A Priori vs. A Posteriori Checking

Now, consider how refactoring tools actually implement refactorings: Preconditions are checked, and if they pass, the transformation is applied... and the refactoring is finished. We will call this *a priori* precondition checking, when preconditions are checked before transformation. An alternative is *a posteriori* precondition checking: checking preconditions *after* the program has been transformed.

To our knowledge, automated refactoring tools use *a priori* precondition checking exclusively in practice. Unfortunately, it has two major disadvantages:

- Since the program is not analyzed after it has been transformed, there are no "sanity checks" on the refactoring... not even a guarantee that the transformed program will parse.
- *A priori* checks make decisions based on what a transformation is *expected* to do. If there is an error in the transformation, or if it is changed at some point in time, the preconditions may not be sufficient.

The case for *a posteriori* checking is bolstered when we consider the three properties that preconditions guarantee. Clearly, input validation needs to be performed *a priori,* since it may not even be possible to perform a transformation if the user provides invalid input. But compilability is actually quite easy to determine *a posteriori;* essentially, it means running the program through a compiler front end. And it turns out that preservation preconditions can often be checked *a posteriori* as well.

3. DIFFERENTIAL REFACTORING ENGINES

The basic distinction between a traditional refactoring engine and a differential refactoring engine is illustrated in Figure 1. A traditional refactoring engine proceeds in three steps (Figure 1(a)):

1. Source code is analyzed, and a program representation is constructed.
2. Preconditions are checked to validate user input and to ensure compilability and behavior preservation.
3. The source code is modified.

In contrast, a differential refactoring engine proceeds in seven steps (Figure 1(b)):

1. Source code is analyzed, and a program representation is constructed.
2. A *semantic model* is constructed from this program representation. This is called the *initial model.*
3. User input is validated.
4. The source code is modified.
5. The modified source code is analyzed, and a new program representation is constructed. Compilability errors are detected.
6. A semantic model is constructed from this new program representation. This is called the *derivative model.*
7. A *preservation analysis* is performed: The derivative model is compared against the initial model. This is used to (approximately) determine whether or not the transformation preserved behavior.

What distinguishes a differential refactoring engine from a traditional refactoring engine is how it ensures compilability and preservation. Compilability is ensured by essentially performing the same checks that a compiler front end would perform. Behavior preservation is ensured by building *semantic models* of the program before and after it is transformed: If the two models are determined to be equivalent, then the transformation is considered to be behavior-preserving.

At first glance, replacing a three-step process with a seven-step process does not appear to simplify things. However, it actually makes individual refactorings much simpler. This is because the compilability and preservation checks are *generic:* They are implemented once, and then reused in every refactoring. In fact, most of the infrastructure needed to implement them is already available in a traditional refactoring tool.

It is virtually impossible to perform any complicated refactorings without a parser, abstract syntax tree (AST),

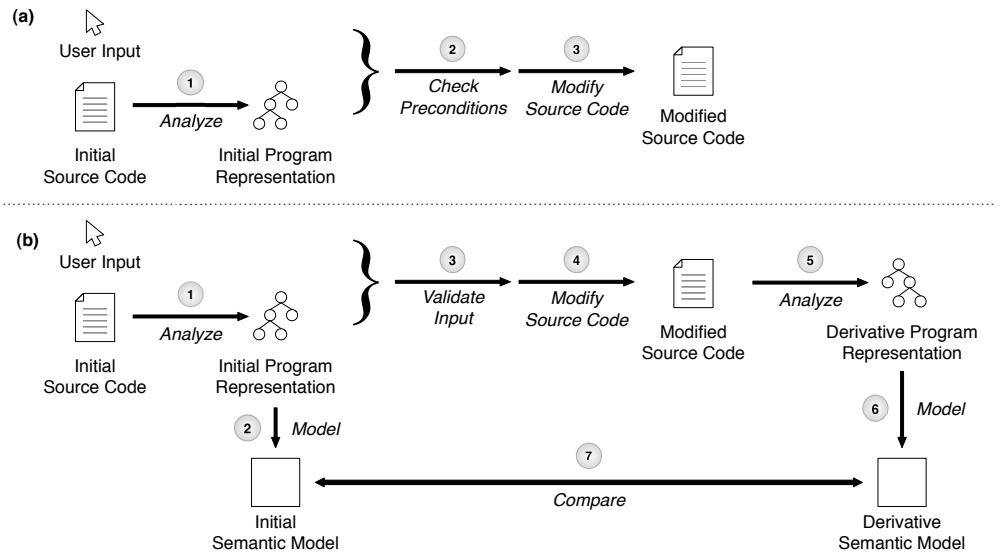


Figure 1: Operation of (a) a traditional refactoring engine and (b) a differential refactoring engine

and name binding information (symbol tables). And a type checker is usually needed to resolve name bindings for members of record types, as well as for refactorings like Extract Local Variable. So, refactoring tools generally contain (most of) a compiler front end. Steps 1 and 5 are simply running source code through this front end. Checking for compilability in Step 5 is natural since a compiler front end would often perform these checks anyway.

Similarly, all refactoring tools require some static analyses, such as name binding and control flow computations. Implementing a differential refactoring engine, then, simply involves reusing these to populate the semantic models.

The only component of a differential refactoring engine that is truly unique is the preservation analysis. Clearly, the most difficult part of building a differential refactoring engine is choosing an appropriate semantic model and finding a preservation analysis algorithm that balances speed, correctness, and generality.

4. PRESERVATION AND REFACTORING

Before attempting to choose a semantic model and design a preservation analysis, it is helpful to consider what, exactly, refactorings try to preserve. The obvious answer is, "Refactorings preserve the program's observable behavior." That is how refactoring is defined in the literature.

In fact, automated refactorings do *not* guarantee behavior preservation for every program. Rather, they make certain assumptions about the program (and the user input) and guarantee behavior preservation only when those assumptions are met. Rename may not preserve behavior in a program that uses reflection. Extract Method may not preserve behavior in a program that accesses its stack trace. When a method is polymorphic, Inline Method may not preserve behavior if the user chooses to inline the "wrong" implementation of the method.

Why is this? Automated refactorings are based on static analyses, and it is often difficult or impossible to determine statically whether these behaviors occur.

So, it is helpful to view "preservation" differently in the context of an automated tool: *An automated refactoring preserves certain aspects of a program analysis.* Rename preserves a name binding relationship: It ensures that every identifier refers to the "same" declaration before and after transformation. Pull Up Method preserves the same relationship while also preserving a relationship between classes and methods they override. Extract Method and Extract Local Variable preserve control flow and def-use chains at the extraction site.

It is important to note that "certain aspects" of a static analysis are preserved, not "all aspects." For example, consider Extract Method: local variables become parameters in the extracted method, so it does not completely preserve name binding relationships. Nor does Encapsulate Variable, which redirects variable accesses through accessor and mutator methods.

In sum, two points should drive our choice of a semantic model and preservation analysis:

1. The semantic model must be able to represent the most common static analyses needed for refactorings: name binding relationships, control flow, and du-chains.
2. The preservation analysis must be able to accommodate or ignore some expected differences between the semantic models.

5. DIFFERENTIAL REFACTORING WITH PROGRAM GRAPHS

In the remainder of this paper, we will discuss a particular type of differential refactoring engine which uses a *program graph* as both its program representation and its semantic model. Although we have found this to be quite satisfactory, it is only one possible semantic model; we encourage investigation of other program representations, semantic models, and preservation analyses. In Section 3, we intentionally made the definition of a differential refactoring engine fairly abstract; the concept is, in general, not limited

to a particular program representation, semantic model, or preservation analysis.

The remainder of this section is organized as follows. Program graphs are described in §5.1. §5.2 describes how program graphs can be used as the program representation for a refactoring tool. §5.3–5.4 describe a preservation analysis on program graphs and how it can be used as the basis of a differential refactoring engine. Safe Delete and Pull Up Method are used as examples in §5.5–5.6.

5.1 Program Graphs

One program representation which has enjoyed success in the refactoring literature [6, 12] is called a program graph. A *program graph* "may be viewed, in broad lines, as an abstract syntax tree augmented by extra edges" [6, p. 253]. These "extra edges"—which we will call *semantic edges*—represent semantic information, such as name bindings, control flow, inheritance relationships, and so forth. Alternatively, one might think of a program graph as an AST with the graph structures of a control flow graph, du-chains, etc. superimposed; the nodes of the AST serve as nodes of the various graph structures.

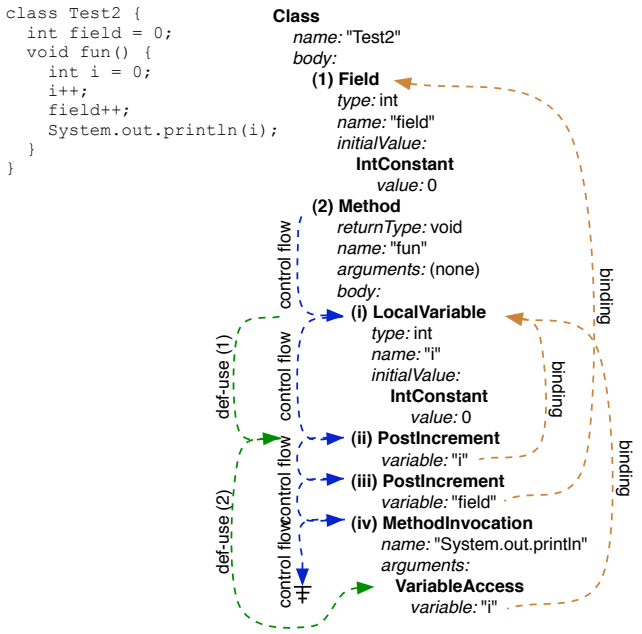


Figure 2: Example Java program and corresponding program graph

An example of a Java program and a plausible program graph representation are shown in Figure 2. The underlying abstract syntax tree is shown in outline form; the dotted lines are the extra edges that make the AST a program graph. We have shown three types of edges. *Name binding* edges link the use of an identifier to its corresponding declaration. Within the method body, *control flow* edges form the (intraprocedural) control flow graph; the method declaration node is used as the entry block and null as the exit block. Similarly, there are two du-chains, given by *def-use* edges.

Program graphs are appealing because they summarize the "interesting" aspects of both the syntax and semantics of a program in a single representation, obviating the need to maintain a mapping between several distinct representations. Moreover, they are defined abstractly: *the definition of a program graph does not state what types of semantic edges are included.* A person designing a program graph is free to include (or exclude) virtually any type of edge imaginable, depending on the language being refactored and needs of the refactorings that will be implemented. We have found five types of edges to be useful: name binding, control flow, du-chains, *override* edges (which link an overriding method to the overridden implementation in a superclass), and *inheritance* edges (which link a class to the concrete methods it inherits from a superclass).

5.2 Program Graphs and AST Manipulation

In the end, refactoring tools manipulate source code. However, when building a refactoring, it is helpful to think of manipulating the AST instead. Adding a node means inserting source code. Replacing a node means replacing part of the source code. And so on.

This does not change when a program graph is used in a refactoring tool. A program graph is always *derived from* an AST. The content of the AST determines what semantic edges will be superimposed. Semantic edges cannot be manipulated directly; they can only change as a side effect of modifying the AST.

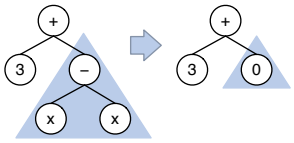
In fact, that observation will serve as the basis of our preservation analysis. When we modify an AST, we will indicate which semantic edges we expect to be preserved and which ones we expect to change. Then, after the source code has been modified, we will determine what semantic edges were actually preserved and compare this with our expectations.

5.3 Preservation in Program Graphs

This raises a question: What does it mean for a semantic edge to be "preserved" when an AST is modified?

We would like to say: If both the modified and unmodified ASTs contain an edge with the same type and the same endpoints, that edge has been preserved. Unfortunately, it is not clear what the "same" endpoints are, since the AST has been modified, and the endpoints are AST nodes.

Consider a refactoring which replaces the expression $x-x$ with the constant 0. When applied to the expression $3+(x-x)$, this corresponds to the following tree transformation.



When a subtree is changed (i.e., added, moved, removed, or replaced) in an AST, we will call that the *affected subtree.* A gray triangle surrounds the affected subtrees in the figure above. Using that figure as an example, consider how AST nodes in the unmodified AST correspond with nodes in the modified AST:

- There is an obvious correspondence between AST nodes outside the affected subtrees, since those parts of the AST were unaffected by the transformation.
- As a whole, the affected subtree before the transformation corresponds to the affected subtree after the

transformation.

- In general, there is no correspondence between nodes inside the affected subtrees.

Recall that our goal is to determine if a semantic edge has the "same" endpoints before and after an AST transformation. Clearly, this is easy when an endpoint is outside the affected subtree, or if that endpoint is the affected subtree itself. But if the endpoint is *inside* the affected subtree, we cannot determine exactly which node it should correspond to... except that, if it corresponds to anything, that node would be in the other affected subtree.

Since we cannot determine a correspondence between AST nodes inside the affected subtree, we will *collapse* the affected subtrees into single nodes. This makes the AST before transformation isomorphic to the AST after transformation.

Now, suppose we have superimposed semantic edges to form a program graph. When we collapse the affected subtree to a single node, we will also need to adjust the endpoints of the semantic edges accordingly:

- When an affected subtree is collapsed to a single node, if any semantic edges have an endpoint inside the affected subtree, that endpoint will instead point to the collapsed node.

Note, in particular, that if an edge has *both* endpoints inside the affected subtree, it will become a self-loop on the collapsed node. Also, note that a program graph is not a multigraph: If several edges have the same types and endpoints in the collapsed graph, they will be merged into a single edge.

Collapsing the affected subtree in a program graph actually has a fairly intuitive interpretation: If we replace one subtree with a different subtree that supposedly does the same thing, then the new subtree should interface with its surroundings in (mostly) the same way that the old subtree did. That is, all of the edges that extended into the old subtree should also extend into the new subtree, and all of the edges that emanated from the old subtree should also emanate from the new subtree. There may be some differences within the affected subtree, but the "interface" with the rest of the AST stays the same.

In some cases, we will find it helpful to replace one subtree with *several* subtrees (or, conversely, to replace several subtrees with one). For example, Encapsulate Variable removes a public variable, replacing it with a private variable, an accessor method, and a mutator method. In these cases, we have an *affected forest* rather than a single affected subtree. The preservation rule is essentially the same: All of subtrees in the affected forest are collapsed into a single unit. When one subtree is replaced with several, this captures the idea that, if an edge extended into the original subtree, then it should extend into one of the subtrees in the affected forest. In the case of Encapsulate Variable, this correctly models the idea that every name binding that pointed to the original (public) variable should, instead, point to either the new (private) variable, the accessor method, or the mutator method.

5.4 Specifying Preservation Requirements

Now that we have established how to determine whether a semantic edge has been preserved across a transformation, we turn to a different question: How can we express which semantic edges we expect to be preserved and which ones we expect to change?

5.4.1 Edge Classifications

From the above description, we can see that whether we want to preserve an edge depends on its type as well as its relationship to the affected subtree. Therefore, it is helpful to classify every semantic edge as either **internal** (both endpoints of the semantic edge occur within the affected subtree), **external** (neither endpoint occurs within the affected subtree), **incoming** (the semantic edge has a source outside the affected subtree and a sink inside it), or **outgoing** (the source is inside the affected subtree, and the sink is outside it).

5.4.2 Notation

Now, we can establish some notation. To indicate what edges we (do not) expect to preserve, we must indicate three things:

1. *The type(s) of edges to preserve.* We will use the letters N, C, D, O, and I to denote name binding, control flow, du-chain, override, and inheritance edges, respectively. (Note, however, that program graphs may contain other types of edges as well, depending on the language being refactored and the requirements of the refactorings being implemented.)
2. *The classification(s) of edges to preserve.* We will use $\leftarrow$, $\rightarrow$, $\circlearrowleft$, and $\times$ to indicate incoming, outgoing, internal, and external edges, respectively. We will use $\leftrightarrow$ as a shorthand for describing both incoming and outgoing edges.
3. *Whether we expect the transformation to introduce additional edges or remove existing edges.* If additional edges may be introduced, we denote this using the symbol $\supseteq$ (i.e., the transformed program will contain a superset of the original edges). If existing edges may be eliminated, we denote this by $\subseteq$. If edges may be both added and removed, then we cannot effectively test for preservation, so those edges will be ignored; we indicate this using the symbol $\neq$. Otherwise, we expect a 1–1 correspondence between edges, i.e., edges should be preserved exactly. We indicate this by $=$.

5.5 Example: Safe Delete (Fortran 95)

To make these ideas more concrete, let us first consider a Safe Delete refactoring for Fortran which deletes an unreferenced internal subprogram.[1]

The traditional version of this refactoring has only one precondition: There must be no references to the subprogram except for recursive references in its definition.

What would the differential version look like? To determine its preservation requirements, it is often useful to fill out a table like the following:

[1] A slightly more complete and much more detailed specification for this refactoring is given in the technical report [9] described in the Evaluation section of this paper.

	N	C	D
$\leftarrow$	$=$	$=$	$=$
$\rightarrow$	$\subseteq$	$=$	$=$
$\circlearrowleft$	$\subseteq$	$\subseteq$	$\subseteq$
$\times$	$=$	$=$	$=$

When a subprogram is deleted, all of the semantic edges inside the deleted subroutine will, of course, disappear, and if the subprogram references any names defined elsewhere (e.g., other subprograms), those edges will disappear. Otherwise, no semantic edges should change.

Notating preservation requirements in tabular form is somewhat space-consuming, since in practice most cells contain $=$. Therefore, we will use a more compact notation. For each edge type, we will use subscripts to indicate which cells are *not* $=$, i.e., what edges should *not* be preserved exactly. For the above table, this would be $N_{\overleftarrow{\subseteq}}^{\circlearrowleft}C_{\subseteq}^{\circlearrowleft}D_{\subseteq}^{\circlearrowleft}$.

So, we can describe the differential version of this refactoring in a single step: *Delete the subprogram definition, ensuring preservation according to the rule* $N_{\overleftarrow{\subseteq}}^{\circlearrowleft}C_{\subseteq}^{\circlearrowleft}D_{\subseteq}^{\circlearrowleft}$.

5.6 Example: Pull Up Method (PHP 5)

For a more interesting example, let us consider a Pull Up Method refactoring for PHP 5, which moves a concrete method definition from a class C into its immediate superclass C'.[2]

5.6.1 Traditional Version

Preconditions.

1. *A method with the same name as M must not already exist in C'.* If M were pulled up, there would be two methods with the same name, or M would need to replace the existing method.
2. *If there are any references to M (excluding recursive references inside M itself), then M must not have private visibility.* If it were moved up, its visibility would need to be increased in order for these references to be preserved.
3. *M must not contain any references to the built-in constants* `self` *or* `__CLASS__`. If it were moved up, these would refer to C' instead of C. (Note that PHP contains both `self` and `$this`: The former refers to the enclosing class, while the latter refers to the *this* object.)
4. *M must not contain any references to private members of C (except for M itself, if it is private).* These would no longer be accessible to M if it were pulled up.
5. *If M overrides another concrete method, no subclasses of C' should inherit the overridden method.* Pulling up M would cause these classes to inherit the pulled up method instead.
6. *The user should be warned if M overrides another concrete method.* If M were pulled up into C', then M would replace the method that C' inherited, changing the behavior of that method in objects of type C', although the user might intend this since he explicitly chose to pull up M into C'.

Transformation. Move M from C to C', replacing all occurrences of `parent` in M with `self`.

[2]Again, a more complete and detailed specification is available [9].

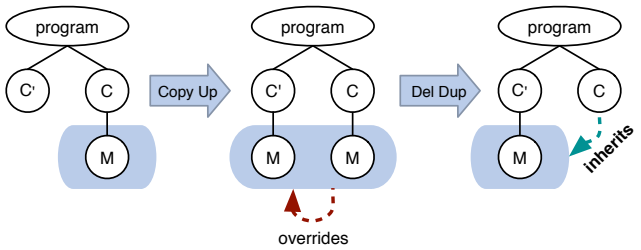
5.6.2 Differential Version

Preconditions. None.

Transformation. The transformation can be expressed as the composition of two smaller refactorings:

1. *Copy Up Method.* Using preservation rule $NO_{\supseteq}^{\circlearrowleft}I$, copy the method definition from C to C', replacing all occurrences of `parent` in M with `self`.
2. *Delete Overriding Duplicate.* Remove the original method definition from C, with rule $NO_{\subseteq}^{\circlearrowleft}I_{\overrightarrow{\supseteq}}^{\circlearrowleft}$.

Pictorially, the process is as follows. The affected forests are highlighted in gray.



When the method is copied from C to C', an internal override edge will be introduced, hence $O_{\supseteq}^{\circlearrowleft}$ in the preservation rule. However, the new method in C' should not be inherited by any subclasses, and all identifiers should bind to the same names they did when the method was contained in C, so the preservation rule is $NO_{\supseteq}^{\circlearrowleft}I$. Once we have established that no subclasses will accidentally inherit the pulled up method, we can delete the original method from C. This will remove the override edge introduced in the previous step, and C will inherit the pulled up method, so the preservation rule is $NO_{\subseteq}^{\circlearrowleft}I_{\overrightarrow{\supseteq}}^{\circlearrowleft}$.

Now, consider how the differential version of this refactoring satisfies all of the traditional version's preconditions. Precondition 1 would be caught by a compilability check. Preconditions 2–4 are simply preserving name bindings. A program that failed Precondition 5 would introduce an incoming inheritance edge. If a program failed Precondition 6, an outgoing inheritance edge from C' would vanish.

For the differential version, we redefined Pull Up Method as the *composition* of two smaller refactorings. Whenever this is possible, it is generally a good idea: It allows preservation rules to be specified at a finer granularity; the smaller refactorings are often useful in their own right; and, perhaps most importantly, simpler refactorings are easier to implement, easier to test, and therefore more likely to be correct.

5.7 On Implementation

While eliminating traditional precondition checks clearly makes refactoring specifications more concise, *it makes refactoring implementations equally concise.* As we will discuss in the next section, we implemented differential refactoring engines in three refactoring tools, including Photran, a refactoring tool for Fortran 95. Figure 3 gives *complete* source code for the Safe Delete refactoring described above—i.e., no additional code is required except for Photran's existing infrastructure. Note that, since the traditional precondition check does not need to be performed—there is no need to explicitly check for references

to the procedure—the entire refactoring consists of just 30 lines of code.

```
public class SafeDeleteRefactoring
        extends PreservationBasedSingleFileFortranRefactoring {
    private ScopingNode nodeToDelete = null;

    protected void doCheckInitialConditions (
        RefactoringStatus status, IProgressMonitor pm)
    {
        ensureProjectHasRefactoringEnabled(status);

        nodeToDelete = findSelected(ScopingNode.class);
        if (!nodeToDelete.isSubprogram())
            fail ("Please select a subprogram to delete.");
    }

    protected PreservationRule [] getEdgesToPreserve() {
        return new PreservationRule [] {
            preserveSubsetOutgoing(BINDING_EDGE_TYPE),
            preserveSubsetInternal (ALL_EDGE_TYPES) };
    }

    protected void doTransform(RefactoringStatus status,
                               IProgressMonitor pm) {
        // Identify the affected subtree ...
        preservation.markDelete( fileInEditor , nodeToDelete);
        // ... then delete it from the AST
        nodeToDelete.removeFromTree();
    }

    public String getName() { return "Safe Delete"; }
}
```

Figure 3: Complete source code for a differential Safe Delete implementation in Photran.

5.8 Summary

To summarize our proposal for a program graph-based differential refactoring engine, we will recast it within the framework of Section 3. Recall from Figure 1(b) that a differential refactoring engine proceeds in seven steps. When our program graph-based approach is employed, these steps are as follows:

1. Source code is analyzed, and a program representation is constructed (generally, AST).
2. The initial model (a program graph) is constructed from this program representation.
3. User input is validated.
4. The source code is modified, and the preservation analysis is notified what rule(s) to use.
5. The modified source code is analyzed, and a new program representation is constructed (generally, an AST). Compilability errors are detected, and the user is notified.
6. The derivative model (a program graph) is constructed from this new program representation.
7. A preservation analysis is performed. The affected subtrees of the two program graphs (the initial model and the derivative model) are collapsed. The semantic edges in the derivative model are compared with the initial model. If they do not satisfy the rules indicated by the refactoring, the user is notified that the transformation is unlikely to preserve behavior.

6. EVALUATION

Again, the primary contribution of this paper is the idea of a differential refactoring engine, the idea that refactorings can be implemented by checking for compilability and preservation *after* transformation in a generic way. Our program graph-based approach demonstrated that this idea can be realized. Our two examples illustrated that it could eliminate explicit precondition checks, simplifying both the specification and implementation of refactorings. It remains to be shown that the program graph-based approach (and, more generally, the differential approach) can be employed effectively to implement a wide range of common refactorings. (Indeed, redesigning a refactoring engine would be of little value if it only improved one or two refactorings.) So we will focus the present evaluation on two questions:

Q1. *Expressivity.* Are the preservation specifications in §5.4 sufficient to implement common automated refactorings?

Q2. *Productivity.* When refactorings are implemented as such, does this actually reduce the number of preconditions that must be explicitly checked?

For our evaluation, we implemented a differential refactoring engine in three refactoring tools: (1) Photran, a popular Eclipse-based IDE and refactoring tool for Fortran; (2) a prototype refactoring tool for PHP 5; and (3) a similar prototype for BC. Implementing a program graph representation and a differential refactoring engine at scale is highly nontrivial. Unfortunately, due to space limitations, a discussion and evaluation of the implementation is beyond the scope of what can be discussed in the present paper; we must reserve this for future work.

6.1 Identifying Common Refactorings

To effectively answer questions Q1 and Q2, we must first identify what the most common automated refactorings are. The best empirical data so far are reported by Murphy-Hill et al. [8], who analyzed the frequency with which various refactorings were used in Eclipse JDT. Table 1 shows several of the top refactorings; the *Eclipse JDT* column shows the popularity of each refactoring according to [8, Table 1, "Everyone"], which represents data voluntarily collected from more than 13,000 developers by the Eclipse Usage Data Collector. For comparison, we have also listed the availability of these refactorings in other popular refactoring tools for various languages.

6.2 Q1: Expressivity

To evaluate the expressivity of our method, we implemented 18 refactorings (see Table 2): 7 for Fortran, 9 for BC, and 4 for PHP. Five of these refactorings are Fortran or BC analogs of the five most frequently-used in Eclipse JDT. Nine others are support refactorings, necessitated by decomposition. The remaining refactorings were chosen for other reasons. Add Empty Subprogram and Safe Delete were the first to be implemented; they helped shape and test our implementation. Introduce Implicit None preserves name bindings in an "interesting" way. Pull Up Method required us to model method overriding and other class hierarchy issues in program graphs.

We divided refactorings among the three languages as follows. We implemented all of the refactorings that rely primarily on name binding preservation in Photran, since

Refactoring	Eclipse JDT (Rank)	IntelliJ IDEA[1]	IntelliJ ReSharper[2]	MS Visual Studio[3]	Eclipse CDT	Visual Assist X[4]	Apple Xcode[5]	Zend Studio[6]
Rename	1	●	●	●	●	●	●	●
Extract Variable	2	●	●	○	●	○	○	●
Move	3	●	●	○	○	○	○	●
Extract Method	4	●	●	●	●	●	●	●
Change Signature	5	●	●	●	○	●	○	○
Pull Up Method	11	●	●	○	○	●	●	○

Legend: ● *Included* ○ *Not Included*

[1] http://www.jetbrains.com/idea/features/refactoring.html
[2] http://www.jetbrains.com/resharper/features/code_refactoring.html
[3] http://msdn.microsoft.com/en-us/library/719exd8s.aspx
[4] http://www.wholetomato.com/products/featureRefactoring.asp
[5] http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/XcodeWorkspace/150-Refactoring/refactoring.html
[6] http://www.zend.com/en/products/studio/features#refactor

Table 1: Automated refactorings in popular tools.

Fortran has the most complicated name binding rules. We implemented flow-based refactorings for BC: It contains functions, scalar and array variables, and all of the usual control flow constructs, but it is a much smaller and simpler language than either Fortran or PHP. This simplified the transformations in these (usually complex) refactorings without sacrificing any of the essential preconditions. We implemented object-oriented refactorings for PHP 5.

We chose to implement 18 refactorings across three very different languages to demonstrate the generality of our technique. A technique that works for 18 refactorings will certainly apply to many others as well. That said, many popular IDEs provide fewer than 10 refactorings, including Apple Xcode (8 refactorings), Microsoft Visual Studio (6), and Zend Studio (4). So while generality is important and desirable, expediting and improving the implementation of a few common refactorings is equally important, perhaps even more so.

6.3 Q2: Productivity

For each of the 18 refactorings, we chose a target language that would provide a challenging yet representative set of preconditions. This brings us to our second research question: Does using a differential engine reduce the number of preconditions that must be explicitly checked? To answer this question, we needed to be able to compare traditional and differential forms of the same refactorings. We also needed to be able to quantify the "amount of precondition checking" required for each refactoring.

Before implementing our refactorings, we wrote detailed specifications, which we have published as a technical report [9]. Each specification describes both the traditional and the differential version of the refactoring, both at a level of detail sufficient to serve as a basis for implementation. (Several undergraduate interns working on Photran implemented refactorings based on our specifications.) The style of the specifications is similar to the Pull Up Method example from §5, except more precise. For example, the Fortran refactoring specifications use the same terminology as the Fortran 95 ISO standard.

We wrote these specifications with the specific intent to provide a "fair" comparison between the traditional and differential versions of the refactorings.[3] We broke down the preconditions for each refactoring into steps, mimicking an imperative implementation, and factored out duplication among refactorings. Assumptions about analysis capabilities were modest—roughly equivalent to a compiler front end coupled with a cross-reference database.

A summary of the refactoring specifications is shown in Table 2. Following the name of the refactoring, the next several columns enumerate all of the preconditions in our specifications and indicate which ones were eliminated by the use of the differential engine. The next two columns attempt to quantify the "amount of precondition checking" involved in each refactoring. A precondition such as "introducing X will preserve name bindings" is far more complicated than a precondition like "X is a valid identifier," so we chose to look at the number of *steps* devoted to precondition checking in the specification of each refactoring. We attempted to make the granularity of each step consistent, so the total number of steps should be a relatively fair measure of the complexity of precondition checking. The *Trad.* column gives the total number of steps in all of the refactoring's precondition checks in the traditional version; the *Diff.* column gives the number of steps in the differential version. These two numbers are also shown as bar graph in Figure 4. For comparison, the last column in the table (*Xform*) gives the number of steps in the transformation; this is not shown in the bar graph.

The data in Table 2 and Figure 4 support our hypothesis that using a differential refactoring engine reduces the amount of explicit precondition checking that must be performed. Notably:

1. *The number of precondition checking steps decreased for most refactorings, often substantially.* When no precondition checks were eliminated, it was generally because the preconditions were not related to compilability or preservation.
2. *The precondition checks that were eliminated tended to be complex,* including a 25-step name binding preservation analysis for Fortran.
3. *The number of precondition checking steps never increased.* In fact, using a differential refactoring engine *cannot* increase the number of preconditions that must be checked. A differential engine provides a "free" check for compilability and preservation preconditions. In the worst case, a refactoring has none of these—in which case, it requires as many precondition checks as it would in a traditional refactoring engine. So, the number of precondition checking steps can only decrease (or stay the same).

6.4 Robustness

Eliminating complex preconditions can improve robustness indirectly by making refactorings simpler to specify and implement. But in fact, *a differential refactoring engine guarantees a certain level of robustness by design.* Checking preservation preconditions *a posteriori* ensures

[3] We originally considered comparing the actual implementations (e.g., by measuring lines of code), but it is well known that such numbers could easily be skewed by details of the implementation not directly attributable to the use of the differential engine, making conclusive results more difficult for the reader to verify.

		Refactoring	Preconditions							Steps		
			IN	SI	PP	RN	II	Other	Warn	Trad.	Diff.	Xform
Fortran	1.	Rename	●	-	-	-	-	-	○	**29**	**1**	4
	2.	Move	-	-	●	●	-	○	-	**23**	**3**	58
	3.	*Introduce* Use	●	-	-	-	-	●	-	**29**	**0**	5
	4.	Change Function Signature	-	-	-	-	-	○	-	**4**	**4**	13
	5.	Introduce Implicit None	-	-	-	-	-	-	-	**0**	**0**	6
	6.	Add Empty Subprogram	●	-	-	-	-	-	-	**27**	**0**	2
	7.	Safe Delete	-	●	-	-	-	-	-	**4**	**0**	5
BC	8.	Extract Local Variable	-	-	-	-	-	●	-	**3**	**0**	4
	9.	*Add Local Variable*	●	-	-	-	-	-	-	**20**	**0**	4
	10.	*Introduce Block*	-	-	-	-	-	-	-	**0**	**0**	1
	11.	*Insert Assignment*	-	-	-	-	-	●	-	**1**	**0**	1
	12.	*Move Expression Into Assignment*	-	-	-	-	-	●	-	**2**	**2**	2
	13.	Extract Function	-	-	-	-	-	○	-	**1**	**1**	4
	14.	*Add Empty Function*	-	-	-	-	-	●	-	**1**	**0**	1
	15.	*Populate Unreferenced Function*	-	-	-	-	-	●	-	**2**	**0**	17
	16.	*Replace Expression*	-	-	-	-	-	-	-	**0**	**0**	13
PHP	17.	Pull Up Method	-	-	-	-	-	-	-	**0**	**0**	2
	18.	*Copy Up Method*	-	-	-	-	●	●	○	**12**	**1**	3

Legend: ● *Eliminated* ○ *Not eliminated* - *Not applicable*

Table 2: Summary of traditional and differential specifications of 18 refactorings. The precondition acronyms and step counts are described in the written specifications [9].

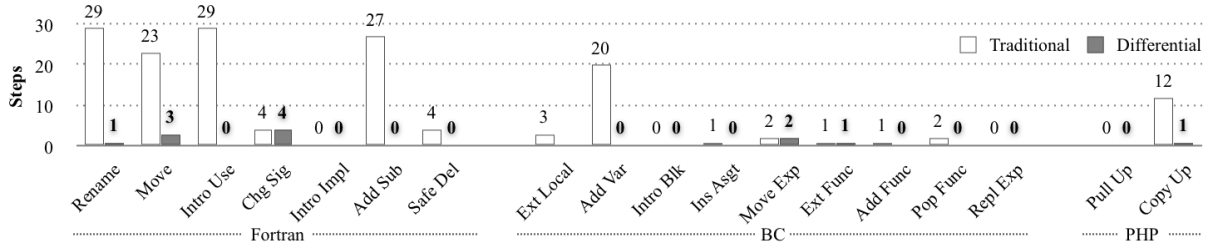


Figure 4: Precondition step counts from Table 2.

that preservation was actually achieved; in the presence of a buggy or incomplete transformation, it analyzes what the transformation *actually did,* not what it was *supposed to do.* Checking for compilability *a posteriori* provides a sanity check: If the code does not compile after refactoring, something must be wrong, and the user should be notified.

The research literature provides particularly compelling evidence that *a posteriori* compilability and preservation checking can improve robustness: These checks, which are done automatically in a differential refactoring engine, subsume some highly nontrivial preconditions—preconditions that developers have "missed" in traditional refactoring implementations. Verbaere et al. [12] identify a bug in several tools' Extract Method refactorings in which the extracted method may return the value of a variable which has not been assigned—a problem which will be identified by a compilability precondition. Schäfer et al. [11] describe a bug in Eclipse JDT's Rename refactoring which amounts to a failure to preserve name bindings. Daniel et al. [1] reported 21 bugs on Eclipse JDT and 24 on NetBeans (many were identical). Of the 21 Eclipse bugs, 19 would have been caught by a compilability check. Seven of these identified missing preconditions;[4] the others were actually errors in the transformation that manifested as compilation errors.

7. LIMITATIONS

Our preservation analysis has two notable limitations.

First, it *assumes* that, if a replacement subtree interfaces with the rest of the AST in an expected way, it is a valid substitute for the original subtree. It is the refactoring developer's responsibility to ensure that this assumption is appropriate. For example, if one were to replace every instance of the constant 0 with the constant 1, this would almost certainly break a program, but our analysis would

[4]Bugs 177636, 194996, 194997, 195002, 195004, 194005, and 195006

not detect any problem, since this change would not affect any edges in a typical program graph. However, the refactoring developer should recognize that name bindings, control flow, and du-chains do not model the conditions under which 1 and 0 are interchangeable values.

Second, for our preservation analysis to be effective, the "behavior" to preserve must be modeled by the program graph. There are several cases where this is unlikely to be true, including the following.

Interprocedural data flow. One particularly insidious example is illustrated by an Eclipse bug (186253) reported by Daniel et al. [1]. In this bug, Encapsulate Field reorders the fields in a class declaration, causing one field to be initialized incorrectly by accessing the value of an uninitialized field via an accessor method. In theory, this could be detected by a preservation analysis, as it is essentially a failure to preserve du-chains for fields among their initializers. Unfortunately, these would probably not be modeled in a program graph, since doing so would require an expensive interprocedural analysis.

Constraint-based refactorings, such as Infer Generics [5]. These refactorings preserve invariants modeled by a system of constraints; a program graph is an unsuitable model.

Library replacements, such as replacing primitive `int` values with `AtomicInteger` objects in Java [2], or converting programs to use `ArrayList` instead of `Vector`. Program graphs generally model *language* semantics, not *library* semantics, and therefore are incapable of expressing the invariants that these refactorings maintain.

8. RELATED & FUTURE WORK

The idea of using *a posteriori* checks in a refactoring tool is new, although it has been hinted at by previous work. In the context of dependence-based transformations, for example, a *fusion preventing dependence* [4, p. 258] is defined in a way such that it is most easily detected *a posteriori*. Roberts [10] first suggested the use of postconditions in refactoring, although his objective was to use them in tandem with *a priori* precondition checks to reduce the cost of implementing composite refactorings: A precondition does not need to be checked if a previous refactoring can guarantee that it has already been satisfied.

Classifying preconditions as guaranteeing *input validity, compilability,* and *preservation* is also new.

The idea that behavior-preserving program transformations maintain invariants is not new, although there is open debate on how, exactly, to express and check these invariants. The notation and technique proposed in §5 is new. Mens et al. [6] used graph rewriting rules. Verbaere et al. [12] used a variant of Datalog. Roberts [10] expressed pre- and postconditions using first-order predicate logic. Griswold [3] proved (by hand) that each transformation's effects on a program dependence graph (PDG) could be described as a composition of meaning-preserving PDG transformations. Schäfer et al. [11] implemented a Rename refactoring for Java by "inverting" name lookup rules, adding qualifiers to names as necessary to guarantee that the name bindings would resolve identically after the transformation was complete—i.e., the name binding invariant was maintained by construction.

Much future work is possible. Many alternatives to our program graph-based semantic model and preservation analysis are possible. In Smalltalk, for example, name binding preservation could be checked using the bytecode representation as a semantic model. Of course, much work remains to be done within the program graph-based framework outlined in §5 as well. The present authors are working on a detailed account of our implementation, including performance measurements. Other refactorings should be investigated: For example, are the preservation specifications in §§5.4–5.5 sufficient to describe dependence-based loop transformations? Can a program graph representation be extended to overcome the limitations outlined in the previous section? Can it model C preprocessor directives? Is it useful to extend a differential refactoring engine with expensive interprocedural analyses for the purposes of testing but to replace these analyses with cheaper, traditional precondition checks in production? Rather than failing when a preservation rule is not met, is it possible to add "repair" strategies to the refactoring engine, similar to the way the JastAddJ rename engine [11] added qualifiers to names? We hope that these, and other questions about differential refactoring, will be investigated in the future.

Portions of this work were supported by the United States Department of Energy under Contract No. DE-FG02-06ER25752. The authors would like to thank Rob Bocchino, John Brant, Brett Daniel, Matthew Fotzler, Ashley Kasza, and Abhishek Sharma.

9. REFERENCES

[1] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *FSE '07*, pages 185–194, 2007.

[2] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *ICSE '09*, pages 397–407, 2009.

[3] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, Washington, 1991.

[4] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, San Francisco, 2002.

[5] A. Kiezun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing Java classes. In *ICSE '07*, pages 437–446, 2007.

[6] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *J. Softw. Maint. Evol.*, 17(4):247–276, 2005.

[7] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Softw.*, 23(4):76–83, 2006.

[8] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE '09*, pages 287–297, 2009.

[9] J. L. Overbey, M. J. Fotzler, A. J. Kasza, and R. E. Johnson. A collection of refactoring specifications for Fortran 95, BC, and PHP 5. Technical Report http://jeff.over.bz/papers/2010/tr-refacs.pdf, 2010.

[10] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, UIUC, 1999.

[11] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for Java. In *OOPSLA '08*, pages 277–294, 2008.

[12] M. Verbaere, A. Payement, and O. de Moor. Scripting refactorings with JunGL. In *OOPSLA '06 Companion*, pages 651–652, 2006.