Differential Precondition Checking:
A Language-Independent, Reusable Analysis for Refactoring Engines

Jeffrey L. Overbey[†], Ralph E. Johnson[‡], and Munawar Hafiz[†]

[†]Department of Computer Science and Software Engineering
Auburn University
{joverbey,munawar}@auburn.edu

[‡]Department of Computer Science
University of Illinois at Urbana-Champaign
johnson@cs.illinois.edu

Abstract

One of the most difficult parts of building automated refactorings is ensuring that they preserve behavior. This paper proposes a new technique to check for behavior preservation; we call this technique *differential precondition checking.* It is simple yet expressive enough to implement the most common refactorings, and the core algorithm runs in linear time. However, the main advantage is that a differential precondition checker can be placed in a library and reused in refactoring tools for many different languages; the core algorithm can be implemented in a way that is completely language independent. We have implemented a differential precondition checker and used it in refactoring tools for Fortran (Photran), PHP, and BC.

1 Introduction

What makes writing a new refactoring tool hard? What are the parts of such a tool? One part is the user interface; refactoring is interactive and requires a good UI. But IDEs like Eclipse provide a good framework for building a UI for a refactoring tool, and most of the UI for a new refactoring tool can be reused from other tools. Another part is the parser and the general language infrastructure. People have tried to reuse the infrastructure from compilers and other tools with mixed results, but our previous work [9] shows that it is possible to generate an infrastructure that is perfectly suited for refactoring, so this is a solved research problem, too. The remaining parts are the refactorings themselves. Automated refactorings have two parts: the transformation—the change made to the user's source code—and a set of *preconditions* which ensure that the transformation will produce a program that compiles and executes with the same behavior as the original program. Authors of refactoring tools agree that precondition checking is much harder than writing the program transformations.

This paper shows how to construct a reusable, generic precondition checker which can be placed in a library and reused in refactoring tools for many different languages. This makes it easier to implement a refactoring tool for a new language.

We call our technique for checking preconditions *differential precondition checking.* A differential precondition checker builds a *semantic model* of the program prior to transformation, simulates the transformation, performs semantic checks on the modified program, computes a semantic model of the modified program, and then looks for differences between the two semantic models. The refactoring indicates what differences are expected; if the actual differences in the semantic models are all expected, then the transformation is considered to be behavior preserving. The changes are applied to the user's code only after the differential precondition checker has determined that the transformation is behavior preserving.

Our technique has several advantages. It is simple, practical, and minimalistic. It does not guarantee soundness, and it is *not* a general method for testing program equivalence. Rather, it is designed to be straightforward, fast, scalable, and just expressive enough to implement preconditions for the most common refactorings. Most importantly, the core algorithm can be implemented in a way that is completely language independent, so it can be optimized, placed in a library, and reused in refactoring tools for many different languages.

To evaluate our approach, we implemented 18 refactorings that used differential precondition checking: 7 for Fortran, 9 for BC, and 2 for PHP. The differential precondition checker is language-independent and was reused among the three refactoring engines. We verified the Fortran refactorings by comparing them with the traditional implementations using the unit tests for the traditional implementations. We could not find comparable refactoring tools for PHP and BC; we tested the new refactorings by porting some relevant unit tests from other languages, including two refactoring benchmarks [10, 11]. We profiled the Rename refactoring on Fortran programs and identified that the main source of overhead is the amount of time taken for the front end to analyze the modified program and recompute the name bindings.

This paper makes five contributions. (Relevant section numbers are noted parenthetically.)

1. It characterizes preconditions as guaranteeing *input validity, compilability,* and *preservation* (§3).
2. It introduces the concept of *differential precondition checking* (§3) and shows how it can simplify precondition checking by eliminating compilability and preservation preconditions (§5).
3. It observes that semantic relationships between the modified and unmodified parts of the program tend to be the most important and, based on this observation, proposes a very concise method for refactorings to specify their preservation requirements (§5).
4. It describes how the main component of a differential precondition checker (called a *preservation analysis)* can be implemented in a way that is both fast and language independent (§7).

5. It provides an evaluation of the technique (§8), considering its successful application to 18 refactorings and its implementation in refactoring tools for Fortran (Photran), PHP, and BC.

This work was originally introduced in ASE '11 [8]. The present article extends that work in three ways: (1) it provides a complete, formal description of the differential precondition checking algorithm (§7.2), (2) performance experiments were conducted on updated hardware with a slightly more optimized implementation (§8.2), and (3) it offers a more complete discussion of trade-offs and implementation issues (§§5.7, 9), based on two years of additional experience using the technique.

2 Precondition Checking

In most tools, each refactoring has its own set of preconditions. These are tested first, and the transformation proceeds only if they pass. Unfortunately, designing a sufficient set of preconditions for a new refactoring is extremely difficult. The author of the refactoring must exhaustively consider every feature in the target language and somehow guarantee that the transformation is incapable of producing an error. 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`.

One promising alternative to traditional precondition checking is to analyze the program *after* it has been transformed, comparing it to the original program to determine whether or not the transformation preserved behavior. This has been used for some dependence-based compiler transformations (e.g., a *fusion preventing dependence* [4, p. 258] is most easily detected after transformation), but researchers have applied it to refactoring tools only recently. Although this technique is not yet used in any commercial tools, research indicates that it tends to make automated refactorings simpler and more robust [13].

So, how can a refactoring tool analyze a program after transformation? Refactorings preserve certain relationships in the source program. The Rename refactoring preserves a name binding relationship: It ensures that every identifier refers to the "same" declaration before and after transformation. Extract Method and Extract Local Variable preserve control flow and def-use chains at the extraction site. As we will see later in this paper, Pull Up Method preserves a name binding relationship, as well as a relationship between classes and methods they override. In our experience, the most common refactorings preserve invariant relationships related to name bindings, inheritance, overriding, control flow, and def-use chains. Analyzing a program after transformation means ensuring that these invariant relationships are preserved across the transformation.

Schäfer et al. have suggested one way to refactor using invariants like these. To implement a Rename refactoring for Java, they stored the original name bindings, changed names, then checked the resulting bindings, adding qualifiers as necessary to guarantee that the name bindings would resolve identically after the transformation was complete [15]. They used a similar approach to implement Extract Method: They stored the original control flow, performed the transformation, then added control flow constructs as necessary to restore the original flow [16]. They have applied this approach to many other refactorings as well [13, 14]. In short, their approach maintains invariants *by construction*—i.e., while performing the transformation, the refactoring checks the invariant and, if possible, adjusts its behavior to preserve it.

The approach taken in this paper is based on some of the same ideas as that of Schäfer et al., but there is a substantial difference in *how* we perform the preservation check. The main difference is that our technique, when implemented appropriately, is language independent; the mechanism for specifying preservation requirements and the algorithm for performing the preservation analysis are the same, regardless of what refactoring is being checked and regardless of what language is being refactored. This means that, unlike the approach of Schäfer et al., our preservation analysis can be implemented in a library and reused verbatim in refactoring tools for many different languages.

3 Differential Precondition Checking

Preconditions determine the conditions under which the program transformation will preserve behavior. Logically, this means that they 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.

Clearly, input validation needs to be performed before the program is transformed, since it may not even be possible to perform a transformation if the user provides invalid input. But compilability is actually easier to determine *after* transformation; essentially, it means running the program through a compiler front end. It turns out that preservation can often be checked *a posteriori* as well.

When *differential precondition checking* is employed, refactorings proceed as follows:

1. Analyze source code and produce a program representation.
2. Construct a *semantic model,* called the *initial model.*
3. Validate user input.

4. Simulate modifying source code, and construct a new program representation. Detect compilability errors, and if appropriate, abandon the refactoring.
5. Construct a semantic model from this new program representation. This is the *derivative model.*
6. Perform a *preservation analysis* by comparing the derivative model with the initial model.
7. If the preservation analysis succeeds, modify the user's source code. Otherwise, abandon the refactoring.

What distinguishes differential precondition checking is how it ensures compilability and preservation. These topics will be discussed in detail in Sections 4 and 5, respectively. It ensures compilability by performing essentially the same checks that a compiler front end would perform. It ensures behavior preservation by building *semantic models* of the program before and after it is transformed. The refactoring informs the differential precondition checker of what kinds of semantic differences are expected; the checker ensures that the actual differences in the semantic models are all expected differences—hence the name *differential* precondition checking.[1]

Note that a differential precondition checker contrasts the program's semantic model *after* transformation with its semantic model *before* transformation. This is different from program metamorphosis systems [12], which provide an "expected" semantic model and then determine whether the transformed program's semantic model is equivalent to the expected model. As we will see in §§5.4–5.6, the mechanism for specifying expected differences in a differential precondition checker is fairly coarse-grained; it does not uniquely characterize the semantics of a particular transformed program but rather identifies, in general, how a refactoring is expected to affect programs' semantics.

4 Checking Compilability

Checking for compilability means ensuring that the refactored program does not contain any syntactic or semantic errors, i.e., that it is a legal program in the target language. These errors would usually be detected by the compiler's front end. Typically, these check constraints like "no two local variables in the same scope shall have the same name" and "a class shall not inherit from itself."

When differential precondition checking is employed, these checks are performed in Step 4 (above), and they are used *in lieu of traditional precondition checks.* For example, a refactoring renaming a local variable A to B would not explicitly test for a conflicting local variable named B; instead, it would simply change the declaration of A to B, and, if this resulted in a conflict, it would be detected by the compilability check.

[1] Why differential "precondition" checking? A refactoring takes user input I and uses it to determine a program transformation $T(I)$. However, a precondition for the *application* of $T(I)$ to the user's source code is that it satisfies the properties of compilability and preservation.

In fact, most refactoring tools already contain most of the infrastructure needed to check for compilability. It is virtually impossible to perform any complicated refactorings without a parser, abstract syntax tree (AST), and name binding information (symbol tables). 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 4 (above) involve running source code through this front end. So checking for compilability in Step 4 is natural.

The literature contains fairly compelling evidence for including a compilability check in a refactoring tool. Compilability checking subsumes some highly nontrivial preconditions—preconditions that developers have "missed" in traditional refactoring implementations. Verbaere et al. [17] 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 check. Schäfer et al. [15] 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. Of the 21 Eclipse bugs, 19 would have been caught by a compilability check. Seven of these identified missing preconditions;[2] the others were actually errors in the transformation that manifested as compilation errors.

Compilability checking also serves as a sanity check. In the presence of a buggy or incomplete transformation, it analyzes what the transformation *actually did,* not what it was *supposed to do.* If the code will not compile after refactoring, the transformation almost certainly did something wrong, and the user should be notified.

5 Checking Preservation

Compilability checking is important but simple. Checking for preservation is more challenging. It involves choosing an appropriate semantic model and finding a preservation analysis algorithm that balances speed, correctness, and generality. In this section, we will use a *program graph* as the semantic model. In Section 7, we will use a slightly different semantic model based on the same ideas.

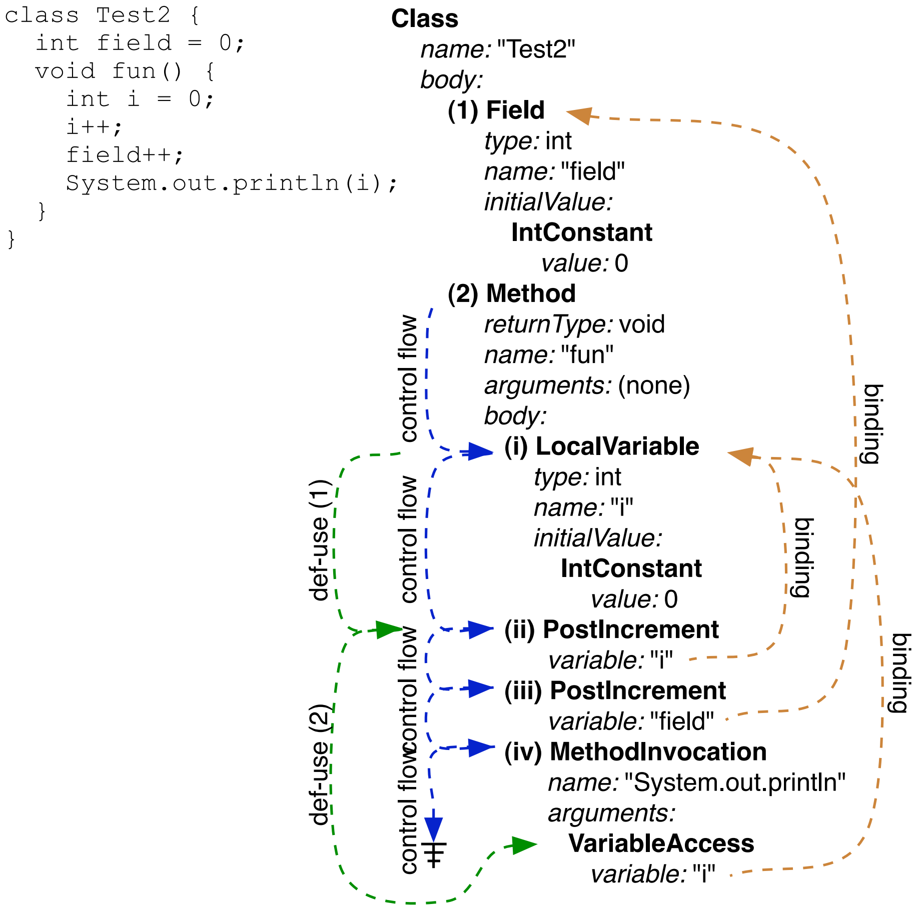
In the remainder of this section, we will discuss what program graphs are (§5.1) and how they can be used as an analysis representation for a refactoring tool (§5.2). Then, we will discuss what *preservation* means in the context of a program graph (§5.3) and how it can be used instead of traditional precondition checks, using Safe Delete and Pull Up Method as examples (§§5.4–5.6).

5.1 Program Graphs

One program representation which has enjoyed success in the refactoring literature [5, 17] is called a program graph. A *program graph* "may be viewed, in

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

broad lines, as an abstract syntax tree augmented by extra edges” [5, 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.



```
class Test2 {
  int field = 0;
  void fun() {
    int i = 0;
    i++;
    field++;
    System.out.println(i);
  }
}
```

Figure 1: Example Java program and corresponding program graph

An example of a Java program and a plausible program graph representation are shown in Figure 1. 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. For the 18 refactorings we considered (see §8), we found five types of edges to be useful: name binding, control flow, def-use, *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.

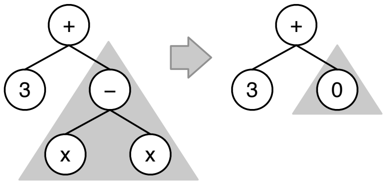
In fact, that observation will serve as the basis of our preservation analysis. When we modify an AST, we will indicate which semantic edges we expect to be preserved and which ones we expect to change. Then, after the source code has been modified, we will determine what semantic edges were actually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

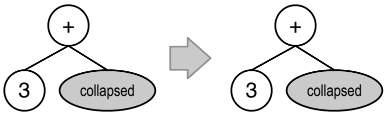


When a subtree is changed (i.e., added, moved, removed, or replaced) in an AST, we will call that the *affected subtree.* A gray triangle surrounds the affected subtrees in the figure above. Using that figure as an example, consider how AST nodes in the unmodified AST correspond with nodes in the modified 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. This is easy if 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 other words, we are modifying several subtrees at the same time. In these cases, we have an *affected forest* rather than a single affected subtree. However, the preservation rule is essentially the same: All of subtrees in the affected forest are collapsed into a single unit. So if an edge extended into some part of the affected forest before transformation, it should also extend into some part of the affected forest after transformation. 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. (We will see an example of an affected forest when we discuss Pull Up Method in §5.6.)

The reader should note that this method of preservation testing does not necessarily guarantee soundness or completeness. We will discuss this further in Section 9. A preservation analysis simply detects how a replacement will change the semantic edges to and from an affected subtree (or forest). The designer of the refactoring must take care to determine edge preservation requirements that adequately model the conditions under which behavior will, in fact, be preserved. As an example, there would usually not be any semantic edges modeling name bindings that occur due to reflection; however, Rename refactorings are known to be unsound in the presence of reflection, so this is a "good enough" approximation.

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 head of the semantic edge is outside the affected subtree but the tail is inside it), or **outgoing** (the head is inside the affected subtree and the tail 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, def-use, 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 $\overset{?}{=}$. 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.[3]

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 (note that Fortran 95 is not object oriented and thus cannot have O- or I-edges):

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

[3] A slightly more complete and much more detailed specification for this refactoring is given in the technical report [7] described in the Evaluation section of this paper.

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. If all cells are =, we will omit the subscript. Using this notation, the preservation requirements in the above table would be notated $N_{\overset{\rightarrow}{\subseteq}\overset{\circlearrowright}{\subseteq}}C_{\overset{\circlearrowright}{\subseteq}}D_{\overset{\circlearrowright}{\subseteq}}$.

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

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'.[4] First, consider the 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).* Private members of C would no longer be accessible to M if it were pulled up.
5. *M must not contain any references to members of C for which there is a similarly-named private member of C'.* These references would refer to the private members of C' if the method were pulled up.
6. *If M overrides another concrete method, no subclasses of C' may inherit the overridden method.* Pulling up M would cause these classes to inherit the pulled up method instead.
7. *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',

[4] Again, a more complete and detailed specification is available [7].

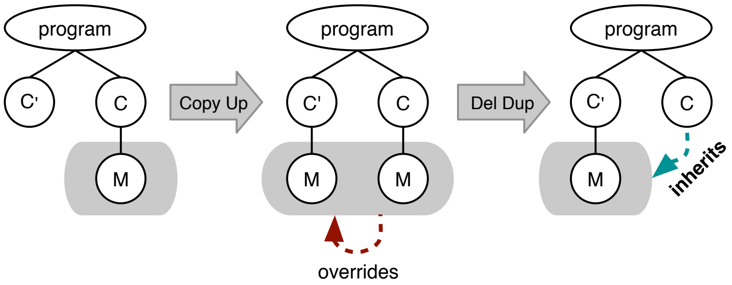
although the user might intend this since he explicitly chose to pull up M into C'.

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

Now, consider the differential version. The transformation can be expressed as the composition of two smaller refactorings:

1. *Copy Up Method.* Using preservation rule $NO\overset{\leftarrow}{\supseteq}\overset{\circlearrowleft}{\supseteq}I\overset{\times}{\subseteq}$, 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 $NO\overset{\circlearrowleft}{\subseteq}I\overset{\leftarrow}{\supseteq}$.

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, as may incoming override edges (if another class will override the pulled up method), hence the rule $O\overset{\leftarrow}{\supseteq}\overset{\circlearrowleft}{\supseteq}$. If the method being pulled up overrides a method inherited from the immediate superclass, then an inheritance edge will be lost, hence $I\overset{\times}{\subseteq}$. 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 no other inheritance or name binding edges are expected to change. 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 $NO\overset{\circlearrowleft}{\subseteq}I\overset{\leftarrow}{\supseteq}$.

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 2–5 are simply preserving name bindings. A program that failed Precondition 6 would introduce an incoming inheritance edge. If a program failed Precondition 7, an outgoing inheritance edge from C' would vanish.

5.7 On Composite Refactorings

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. This has been suggested by other researchers as well [13,16].

However, composing large refactorings from smaller refactorings has one major disadvantage: it can make error reporting difficult. As an example, we have recently begun applying this technique in OpenRefactory/C, a refactoring tool for C; due to various complications in the C language (notably, pointers), the Extract Function refactoring is actually a composition of 11 smaller refactorings. Of course, this is transparent to the user; it appears to be a single, monolithic refactoring. If the preconditions do not pass for one of the intermediate steps, it can be difficult to describe the error in a way that is comprehensible to the end user, since the code is in neither its original form nor its fully refactored form when the precondition failure occurs.

There are several ways to address this problem. One is to provide a custom error handler for each step, which intercepts precondition failures and attempts to display them in a user-friendly form specific to the refactoring. If the error involves a construct in the original source code, the sequence of replacements presented in Section 7.2 can be used to map the construct back to its location in the original source code. Another option—which is not always possible—is to simply let the refactoring continue. If it is known that the subsequent refactoring steps will be able to complete, even in the presence of the error, and the error will still exist in the final, refactored code, then it can be displayed to the user at that point. (We have found that approach to be helpful anecdotally: show what the refactored program would look like, along with error messages pointing to the problematic lines of code.) Of course, a third (less appealing) option is to add explicit precondition checks at the beginning of the composite refactoring and then use the intermediate, differential precondition checks only as a sanity check for the explicit preconditions.

6 The Preservation Analysis Algorithm

If one understands what a program graph is, and what the preservation rules mean, the preservation analysis algorithm is straightforward. A program graph becomes an abstract data type with

Sorts: ProgramGraph, Edge, Type
Operations:

$$\begin{aligned} \text{getAllEdges} &: \text{ProgramGraph} \to \text{finite set of Edge} \\ \text{classify} &: \text{Edge} \to \{\leftarrow, \rightarrow, \circlearrowleft, \times\} \\ \text{type} &: \text{Edge} \to \text{Type} \\ \text{equiv} &: \text{Edge} \times \text{Edge} \to \{\textsc{true}, \textsc{false}\}. \end{aligned}$$

The *equiv* operation determines whether two edges—one in the original program graph and one in the transformed program graph—are equivalent, i.e., if the edge was preserved. For simplicity, we have left this underspecified, although its intent should be clear from the previous section. Now, preservation is determined by the following algorithm.

```
Input:   $P$ : ProgramGraph     *(Original program)*
         $P'$ : ProgramGraph    *(Transformed program)*
         $rule$ : Type $\times$ $\{\leftarrow, \rightarrow, \circlearrowleft, \times\} \rightarrow \{=, \subseteq, \supseteq, \overset{?}{\neq}\}$
Output: PASS or FAIL
let $E \coloneqq \mathbf{getAllEdges}(P)$
let $E' \coloneqq \mathbf{getAllEdges}(P')$
for each Edge $e \in E$
   if $rule(\mathbf{type}(e), \mathbf{classify}(e))$ is $\supseteq$ or $=$
      but $\nexists\ e' \in E'$ s.t. $\mathbf{equiv}(e, e') =$ TRUE, then
         FAIL
for each Edge $e' \in E'$
   if $\mathbf{rule}(\mathbf{type}(e'), \mathbf{classify}(e'))$ is $\subseteq$ or $=$
      but $\nexists\ e \in E$ s.t. $\mathbf{equiv}(e, e') =$ TRUE, then
         FAIL
otherwise, PASS
```

In short, E consists of all of the semantic edges from the original program, and E' consists of all of the semantic edges from the transformed program. The first *for* loop checks for edges in the original program that are missing from the transformed program, failing if it finds one that should have been preserved but was not. The second *for* loop checks for edges in the transformed program that were not present in the original program, failing if if finds an edge that should not have been introduced.

7 Analysis with Textual Intervals

7.1 Overview

The key to an efficient implementation is being able to determine, for a particular edge, whether an equivalent edge exists in the transformed program. If this can be done in $O(1)$ time, then the above algorithm's execution time is linear in the number of edges in the two program graphs. In this section, we will sketch one way to do this (which also makes the implementation language independent).

The ASTs in refactoring tools tend to model source code very closely. This means that they tend to exhibit a very useful property: Every node in an AST corresponds to a particular textual region of the source code, *and this textual region can be mapped back to a unique AST node.* Consider the program graph from Figure 1. The source code is 115 characters long. The *Class* AST node corresponds to the entire source code—the characters at offsets 0 through 114, inclusive, or the interval $[0, 114]$. The field declaration `int field = 0;` corresponds to the interval $[14, 30]$. The post-increment `field++;` becomes

Initial Model (a)

[74, 78]	$\triangleright_N$	[14, 30]
[65, 65]	$\triangleright_N$	[46, 60]
[106, 106]	$\triangleright_N$	[46, 60]
[31, 113]	$\triangleright_C$	[46, 60]
[46, 60]	$\triangleright_C$	[61, 69]
[61, 69]	$\triangleright_C$	[70, 82]
[70, 82]	$\triangleright_C$	[83, 109]
[83, 109]	$\triangleright_C$	[−1, −1]
[46, 60]	$\triangleright_D$	[61, 69]
[61, 69]	$\triangleright_D$	[106, 106]

Norm. Initial (b)

$*$	$\triangleright_N$	$*$
[61, 61]	$\triangleright_N$	[42, 56]
[98, 98]	$\triangleright_N$	[42, 56]
[27, 105]	$\triangleright_C$	[42, 56]
[42, 56]	$\triangleright_C$	[57, 65]
[57, 65]	$\triangleright_C$	$*$
$*$	$\triangleright_C$	[75, 101]
[75, 101]	$\triangleright_C$	[−1, −1]
[42, 56]	$\triangleright_D$	[57, 65]
[57, 65]	$\triangleright_D$	[98, 98]

Norm. Deriv. (c)

[61, 61]	$\triangleright_N$	[42, 56]
$*$	$\triangleright_N$	[42, 56]
[98, 98]	$\triangleright_N$	[42, 56]
[27, 105]	$\triangleright_C$	[42, 56]
[42, 56]	$\triangleright_C$	[57, 65]
[57, 65]	$\triangleright_C$	$*$
$*$	$\triangleright_C$	[75, 101]
[75, 101]	$\triangleright_C$	[−1, −1]
[42, 56]	$\triangleright_D$	[57, 65]
[57, 65]	$\triangleright_D$	[66, 74]
$*$	$\triangleright_D$	[98, 98]

Deriv. Model (d)

[61, 61]	$\triangleright_N$	[42, 56]
[70, 70]	$\triangleright_N$	[42, 56]
[98, 98]	$\triangleright_N$	[42, 56]
[27, 105]	$\triangleright_C$	[42, 56]
[42, 56]	$\triangleright_C$	[57, 65]
[57, 65]	$\triangleright_C$	[66, 74]
[66, 74]	$\triangleright_C$	[75, 101]
[75, 101]	$\triangleright_C$	[−1, −1]
[42, 56]	$\triangleright_D$	[57, 65]
[57, 65]	$\triangleright_D$	[66, 74]
[66, 74]	$\triangleright_D$	[98, 98]

Figure 2: Textual interval models of the program graph from Figure 1, when `field` is renamed to `i`

$[70, 82]$.

Since AST nodes can be represented as intervals, we can use these intervals to describe the semantic edges of a program graph. For example, the name binding edge from the post-increment to the field declaration becomes $[70, 82] \triangleright_B [14, 30]$. (The interval representation of the program graph in Figure 1 is shown in Figure 2(a).)

During a refactoring transformation, it is possible to track what regions of the original source code are deleted or replaced, as well as where new source code is inserted. These textual regions are contained in the affected forests. Since we know exactly how many characters were added or deleted at what positions, then for any character *outside* these regions, it is possible to determine exactly where that character should occur in the transformed program. Suppose we have a (partial) function $newOffset(n)$ that can determine this value, for a given character offset n in the original program.

Now, suppose we take each edge of the derivative model, and if an endpoint is contained in the affected forest, we replace that interval with $*$. We will call the result the *normalized derivative model.* Then, we can take each edge of the initial program graph and use the *newOffset* function to determine the equivalent edge in the normalized derivative model, likewise replacing endpoints in the affected forest with $*$. We will call this the *normalized initial model.*

If the normalized models are stored as sets (eliminating duplicate edges), then each edge in the initial model corresponds to exactly one edge in the normalized initial model, and each edge in the derivative model corresponds to exactly one edge in the normalized derivative model. Now, an edge in the initial model is equivalent to an edge in the derivative model (in the notation of the previous section, $\mathit{equiv}(e, e')$ if, and only if, their corresponding edges in the normalized models are *equal.* By storing the edges of the normalized models in appropriate data structures (e.g., hash sets), we can determine in $O(1)$ time if a particular edge occurs in either model.

An example is shown in Figure 2. Suppose, in the Java program in Figure 1, we attempt to rename the field declaration from `field` to `i`. The transformation is simple: replace the five characters `field` at offsets 20–24 (the declaration) and 74–78 (the reference) with the one-character string `i`. Since four characters are deleted in each case,

$$\mathit{newOffset}(n) = \begin{cases} n & \text{if } n \leq 19 \\ n-4 & \text{if } 25 \leq n \leq 73 \\ n-8 & \text{if } 79 \leq n. \end{cases}$$

The affected forest consists of the field declaration and the second post-increment (initial intervals $[14, 30]$ and $[70, 82]$, derivative intervals $[14, 26]$ and $[66, 74]$). Since `field++` changes to `i++`, the name binding edge for the field reference disappears and becomes a reference to the local variable `i` in the derivative model. Also, a new def-use chain is introduced. Since the renaming transformation would not preserve name bindings (or du-chains, for that matter), it should not be allowed to proceed.

Implementing the preservation analysis using textual intervals, rather than directly on the program graph, has a number of advantages. It allows the preservation analysis to be highly decoupled from the refactoring tool's program representation, which makes it more easily reusable. It is fairly space-efficient, since semantic edges are represented as tuples of integers. Also, there is a fairly natural way to display errors: highlight the affected region(s) of the source code.

7.2 Detailed Construction

We will now turn to the details of implementing a textual interval-based preservation analysis. (Readers uninterested in these details may skip ahead to Section 8.)

We begin with some preliminary definitions. We will denote textual regions using *right half-open integer intervals.* Using half-open intervals allows for many different empty intervals; e.g., $[3, 3)$ denotes an empty interval at position 3, while $[5, 5)$ denotes an empty interval at position 5. This will become important momentarily when we introduce *replacements.*

Definition 1. *A* ***right half-open interval over*** $\mathbb{Z}$ *(or simply "interval") is an ordered pair denoted*

$$I = [\underline{I}, \overline{I}),$$

where $\underline{I}, \overline{I} \in \mathbb{Z}$ and $\underline{I} \leq \overline{I}$. $\underline{I}$ is called the ***lower bound*** *of I, and $\overline{I}$ is called the* ***upper bound*** *of I. The set of all such intervals will be denoted $\mathbb{IZ}$. An interval $[\underline{I}, \overline{I})$ intuitively corresponds to the set $[\![I \,)\!) := \{\underline{I}, \underline{I}+1, \ldots, \overline{I}-1\}$, so we will adopt the following notations from set theory. Let $n \in \mathbb{Z}$.*

- *$n \in I$ denotes $\underline{I} \leq n < \overline{I}$.*
- *$I \subseteq J$ denotes $\underline{J} \leq \underline{I} \leq \overline{I} \leq \overline{J}$.*
- *$I \subset J$ denotes $I \subseteq J \wedge I \neq J$.*
- *$|\, I \,|$ denotes $\max(\overline{I} - \underline{I}, 0)$.*
- *$I \cap J$ denotes the set $\{\max(\underline{I}, \underline{J}), \max(\underline{I}, \underline{J})+1, \ldots, \min(\overline{I}-1, \overline{J}-1)\}$.*

(Note that $I \cap J = \varnothing$ if I and J do not overlap.)

As stated earlier, a textual interval model requires that every node in an AST be mapped to a unique textual region of the source code, *and that this textual region be mapped back to a unique AST node.* Formally, this means that there must be a *textual mapping* defined on the tree, as follows.[5]

Definition 2. *For a directed tree T with vertex set V, a* ***textual mapping***

$$rgn : V \xrightarrow{1\text{–}1} \mathbb{IZ}$$

is an injective (1–1) function with the following properties, for $v, u \in V$.

1. *If u is a descendent of v in T, then $rgn(u) \subset rgn(v)$.*
2. *If u is a sibling of v in T, then $rgn(u) \cap rgn(v) = \varnothing$.*

7.2.1 Predicting Offsets

When the AST is modified during a refactoring transformation, the modified subtrees of the AST comprise the affected forest. The textual mapping allows these modified subtrees to be mapped to textual regions. So, it is possible to track what regions of the original source code are deleted or replaced, as well as where new source code is inserted, based on the changes made to the AST.

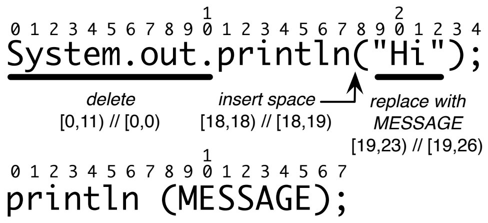
In the author's implementation, this is accomplished using the Observer pattern [3]: The preservation analysis registers an observer on the relevant AST(s), so it can be informed when AST nodes are added, modified, or deleted. The observer uses the textual mapping to map the changed part(s) of the AST to textual regions.

In a textual interval-based analysis, the changes made to the source code are represented as a set of *nonoverlapping replacements.* Each replacement describes an AST node (equivalently, a region of source code) that was added, deleted, or modified.

Definition 3. *A* ***replacement*** *is an ordered pair denoted by $J /\!\!/ K$, where $J, K \in \mathbb{IZ}$ and $\underline{J} = \underline{K}$. We will let R denote the set of all replacements.*

[5] Actually, in practice, the requirement is not so strict: Two tree nodes can correspond to the same textual region as long as only one of them can ever occur as the endpoint of a semantic edge in a program graph.

For example, the string "`System.out.println("Hi");`" can be transformed into the string "`println (MESSAGE);`" using three replacements, as shown below.



Definition 4. *A set $S \subseteq R$ of replacements is* ***nonoverlapping*** *iff*

$$\bigcap_{J /\!\!/ K \in S} J = \varnothing.$$

Determining whether two intervals overlap is quite simple, due to the following theorem, which is proved easily.

Theorem 1. *(Efficient Computation of Interval Overlap)*
Given $I, J \in \mathbb{IZ}$,

$$[\![I)\!) \cap [\![J)\!) = \varnothing \text{ iff } (\overline{I} \leq \underline{J} \vee \underline{I} \geq \overline{J}). \quad \square$$

Since we know exactly how many characters were added or deleted at what positions, then for any character *outside* these regions, it is possible to determine exactly where that character should occur in the transformed program.

Definition 5. *Given $n \in \mathbb{Z}$ and a set S of nonoverlapping replacements, the* ***new offset of*** *n (according to S) is given by the function*

$$\mathit{newOffset}_S(n) := n + \sum_{J /\!\!/ K \in S} \mathit{adjust}_{J /\!\!/ K}(n)$$

where

$$\mathit{adjust}_{J /\!\!/ K}(n) := \begin{cases} 0 & \text{if } n < \overline{J} \\ |K| - |J| & \text{if } n \geq \overline{J}. \end{cases}$$

Intuitively, any single replacement $J /\!\!/ K$ inserts $|K| - |J|$ characters, so all of the characters after the affected interval will be shifted by that many characters. The summation simply computes the total amount by which n will be adjusted after every replacement has been applied. In the example above, the final semicolon was at offset 24 in the original string and offset 17 in the modified string. This is predicted by the newOffset function as follows. The set S consists of three nonoverlapping replacements:

$$S := \{ \quad [0, 11) /\!\!/ [0, 0), \quad [18, 18) /\!\!/ [18, 19), \quad [19, 23) /\!\!/ [19, 26) \quad \}.$$

Then, we have

$$\begin{aligned}
\mathrm{adjust}_{[0,11)/\!\!/[0,0)} &= -11 \\
\mathrm{adjust}_{[18,18)/\!\!/[18,19)} &= 1 \\
\mathrm{adjust}_{[19,23)/\!\!/[19,26)} &= 3 \\
\sum_{J/\!\!/K\in S} \mathrm{adjust}_{J/\!\!/K}(24) &= -7 \\
\mathrm{newOffset}_S(24) &= 24 + -7 = 17.
\end{aligned}$$

7.2.2 Interval Models

Thus far, we have seen that a textual mapping is used to map between AST nodes and textual intervals; the affected forest can be represented as a set of nonoverlapping replacements; and the newOffset function can determine, for any character occurring outside the affected forest, where that character will be located in the transformed program. Now, we will show how to construct semantic models based on these results.

Since an interval uniquely determines an AST node, a semantic edge in the AST can be represented as an ordered triple consisting of (1) the interval corresponding to the head AST node, (2) the edge type, and (3) the interval corresponding to the tail AST node. An *interval model*, then, is simply the set of all semantic edges in a program graph.

Definition 6. *Given a set Σ_E of edge types, an* ***interval model*** *is a finite subset of $\mathbb{IZ} \times \Sigma_E \times \mathbb{IZ}$. The members of this set are called* ***(semantic) edges.*** *An edge (I, ℓ, J) will be denoted by $I \triangleright_\ell J$.*

The **initial model** is the interval model constructed from the original source code, and the **derivative model** is the interval model constructed from the modified source code. To check for preservation, we must construct *normalized* initial and derivative models. This means that we must be able to determine whether an endpoint of an edge lies within the affected forest so that we can replace that endpoint with $*$.

Recall that the affected forest is denoted by a set of nonoverlapping replacements. For a replacement $J/\!\!/K$, the interval J describes the offsets within the original source code that are affected.

Definition 7. *The* ***affected initial interval*** *of a replacement $J/\!\!/K$ is given by*

$$aii(J/\!\!/K) := J.$$

Now, we can determine what part(s) of the *modified* source code lie within the affected forest using the newOffset function.

Definition 8. *Let S be a set of nonoverlapping replacements. The* ***affected derivative interval*** *of a replacement $J/\!\!/K \in S$ is given by*

$$adi_S(J/\!\!/K) := \Big[newOffset_{S-\{J/\!\!/K\}}(\underline{K}), newOffset_{S-\{J/\!\!/K\}}(\underline{K}) + |K|\Big).$$

We can collapse the edges in the derivative model to construct the normalized derivative model.

Definition 9. *Given a set S of nonoverlapping replacements and an interval model D (the* derivative model*), the* ***normalized derivative model*** *is the interval model*

$$\mathit{dnorm}_S(D) := \{\mathit{collapse}_S(I) \rhd_t \mathit{collapse}_S(J) \mid I \rhd_t J \in D\}$$

where

$$\mathit{collapse}_S(I) := \begin{cases} * & \textit{if } \exists r \in S.\ I \subseteq \mathit{adi}_S(r) \\ I & \textit{otherwise.} \end{cases}$$

Constructing the normalized initial model is slightly more difficult. We know what regions of the original source code (and, thus, what AST nodes) lie within the affected forest, and for any character *outside* these regions, we have a function to determine exactly where that character should occur in the transformed program. So, for any interval in the original program, we can predict what the equivalent interval will be in the normalized derivative model: If it lies within the affected forest, it will be $*$; otherwise, we can determine the exact bounds using the newOffset function.

Definition 10. *Given a set S of nonoverlapping replacements and an interval model I (the* initial model*), the* ***normalized initial model*** *is the interval model*

$$\mathit{inorm}_S(I) := \{\mathit{predict}_S(I) \rhd_t \mathit{predict}_S(J) \mid I \rhd_t J \in D\}.$$

where

$$\mathit{predict}_S(I) := \begin{cases} * & \textit{if } \exists J /\!\!/ K \in S.\ I \subseteq \mathit{aii}(J /\!\!/ K) \\ \big[\mathit{newOffset}_S(\underline{I}), \mathit{newOffset}_S(\overline{I} - 1) + 1\big) & \textit{otherwise.} \end{cases}$$

Perhaps the most surprising part of the above definition is the presence of $\cdots - 1) + 1$. This ensures that, if $\overline{I}$ is the start of the affected derivative interval, it is kept alone and not extended to the right side of the interval. For example, suppose a statement S covers offsets 10–20, inclusive; this would be represented by the interval $[10, 21)$. If a new statement is inserted after S, this should not change S's textual interval; it should still be $[10, 21)$. However, suppose S (and the new statement) are contained in a compound statement covering the interval $[9, 22)$. Then the addition of the new statement *should* change the textual interval for the compound statement, since the new statement was added to it.

7.2.3 Performing the Preservation Analysis

Once the normalized initial and derivative models have been constructed according to the above definitions, the preservation analysis is a straightforward implementation of the algorithm from Section 6. Let E denote the normalized initial model (i.e., a set of ordered triples as defined in Definition 10) and E' the normalized derivative model (Definition 9). Define

$$\begin{array}{rcl}\mathbf{type}(v \triangleright_t u) & \coloneqq & t \\ \mathbf{classify}(v \triangleright_\ell u) & \coloneqq & \begin{cases} \leftarrow & \text{if } v \neq * \wedge u = * \\ \rightarrow & \text{if } v = * \wedge u \neq * \\ \circlearrowleft & \text{if } v = * \wedge u = * \\ \times & \text{if } v \neq * \wedge u \neq *. \end{cases}\end{array}$$

Then the preservation analysis exactly follows the pseudocode given in Section 6.

7.2.4 Summary

In sum, a differential precondition checker based on interval models operates as follows.

1. Analyze source code and produce an AST.
2. Construct the initial model I from the AST, performing any requisite static analyses.
3. Validate user input.
4. Perform the transformation, recording the AST changes as a set of nonoverlapping replacements S.
5. Detect compilability errors, and if appropriate, abandon the refactoring.
6. Construct the derivative model D from an AST for the modified source code, again performing any requisite static analyses.
7. Construct the normalized initial model $\mathrm{inorm}_S(I)$ and the normalized derivative model $\mathrm{dnorm}_S(D)$.
8. Apply the preservation analysis algorithm as described in Section 6.
9. If the preservation analysis succeeds, modify the user's source code. Otherwise, abandon the refactoring.

8 Evaluation

In previous sections, we illustrated differential precondition checking using Safe Delete, Pull Up Method, and Rename as illustrative examples. We also sketched a linear-time algorithm for performing the preservation analysis and argued for its language independence. But is this technique effective in practice? We will focus on two questions:

Q1. *Expressivity.* Are the preservation specifications in §3 sufficient to implement the most common automated refactorings?

Q2. *Performance.* When preconditions are checked differentially, what are the performance bottlenecks? How does the performance compare to a traditional implementation?

For our evaluation, we implemented a differential precondition checker which we reused 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.

8.1 Q1: Expressivity

To effectively answer question Q1, we must first identify what the most common automated refactorings are. The best empirical data so far are reported by Murphy-Hill et al. [6]. Table 1 shows several of the top refactorings; the *Eclipse JDT* column shows the popularity of each refactoring in the Eclipse JDT according to [6, Table 1, "Everyone"]. For comparison, we have also listed the availability of these refactorings in other popular refactoring tools for various languages.

Table 1: Automated refactorings in popular tools (as reported in [8]).

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

We selected 18 refactorings (see Table 2): 7 for Fortran, 9 for BC, and 2 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.

It is worth noting that 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 (certainly, a

Table 2: Refactorings evaluated (as reported in [8]).

Fortran	1. Rename	BC	10. Extract Local Variable
	2. Move		11. *Add Local Variable*
	3. *Introduce Use*		12. *Introduce Block*
	4. Change Function Signature		13. *Insert Assignment*
	5. Introduce Implicit None		14. *Move Expression*
	6. Add Empty Subprogram		15. Extract Function
	7. Safe Delete		16. *Add Empty Function*
PHP	8. Pull Up Method		17. *Populate Function*
	9. *Copy Up Method*		18. *Replace Expression*

technique that works for 18 refactorings will apply to many others), expediting and improving the implementation of a few common refactorings is equally important, perhaps more so.

We wrote detailed specifications of all 18 refactorings in a technical report [7]. 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 §3 but more precise. For example, the Fortran refactoring specifications use the same terminology as the Fortran 95 ISO standard.

We divided refactorings among the three languages as follows. For all of the refactorings that rely primarily on name binding preservation, we targeted Fortran, since it has the most complicated name binding rules. We targeted 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 kept the specifications of these (usually complex) refactorings to a manageable size without sacrificing any of the essential preconditions. The one object-oriented refactoring targeted PHP 5.

We implemented a differential precondition checker (following §7) and used it to implement differential refactorings in the three refactoring tools, following our detailed specifications. For BC and PHP, we implemented refactorings as listed in Table 2. Since there are no comparable refactoring tools for these languages, we could not perform differential testing. However, we ported several relevant unit tests from the Eclipse CDT and JDT, as well as two informal refactoring benchmarks [10, 11]. For Fortran, we implemented differential versions of Rename, Introduce Implicit None, Add Empty Subprogram, and Safe Delete. Photran included traditional versions of these refactorings, with fairly extensive unit tests, so we were able to reuse the existing test cases to test the differential implementations.

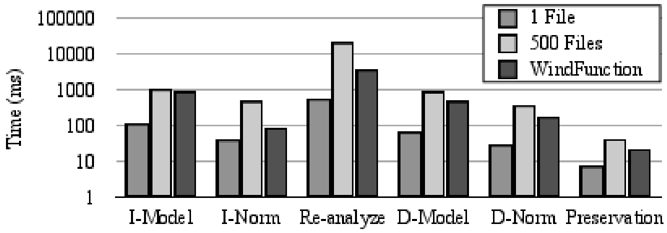


Figure 3: Rename performance profile

8.2 Q2: Performance

Since a differential precondition checker's performance depends on the speed of the language-specific front end, as well what refactoring is being performed and what program is being refactored, it is difficult to make any broad claims about performance. In our experience, when a refactoring affects only one or two files in a typical application, the amount of time devoted to precondition checking is negligible. Most of the refactorings we implemented fall into this category. Performance becomes a concern only at scale, e.g., when a refactoring potentially affects every file in a project. We will use Photran's Rename refactoring as an illustrative example. Rename is the most expensive of the refactorings we implemented, since it can potentially change name bindings in every file in the program, it often makes many changes to a single file, and computing name bindings involves accessing a index/cross-reference database.

Figure 3 shows a performance profile[6] for the Rename refactoring on three Fortran programs. Two are examples intended to test scalability: "1 File" is a project with 500 subroutine definitions in a single file, while "500 Files" contains 1 subroutine in each of 500 files. "WindFunction" shows the results of renaming of the wind function in an atmospheric dispersion simulation (a production Fortran program consisting of about 53,000 LOC in 29 files, four of which were ultimately affected by the refactoring). From left to right, the performance measurements represent creation of the initial interval model, normalization of this model, running the front end to re-analyze the modified code, construction of the derivative interval model, normalization of this model, and, finally, the preservation analysis.

These performance tests were previously performed for the conference version of this work [8]. While the times reported in Figure 3 are slighly better, due

[6] The tests were performed on a 2.3 GHz Intel Core i7 (MacBook Pro), Java 1.6.0_51, with the JVM heap limited to 512 MB.

to better hardware and a slightly more optimized implementation, the overall performance profile remains the same.

Note the logarithmic scale on the y-axis: In all three cases, the performance bottleneck was, by far, the *Re-analyze* measurement—i.e., the amount of time taken for the front end to analyze the modified program and recompute name bindings. This was generally true for other refactorings as well. It is not particularly surprising: When an identifier in one file can refer to an entity in another file, computing name bindings involves populating and accessing a cross-reference database.

9 Limitations

Our preservation analysis has three 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, replacing every instance of the constant 0 with the constant 1 would almost certainly break a program, but our analysis would 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 interprocedural analysis.

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

Complex transformations whose correctness cannot be expressed by syntactic invariants. These include transformations such as loop unrolling, converting programs with GOTO statements to structured programs, and converting programs to static single-assignment form. The preservation specifications described in this paper are only intended to express simple, syntactic relationships in the source code.

This is actually a limitation by design: the types of preservation requirements that can be specified using the rules described earlier is intentionally limited, due to the fact that our technique collapses the entire affected forest into a single unit in the program graph. This was done because (1) we found it to be effective in the common case, as noted in the Evaluation (§8), and (2) it kept the language for specifying preservation requirements relatively simple.

One obvious modification to our approach would allow different subtrees in the affected forest to be distinguished. For example, in the Copy Up Method refactoring for PHP (described earlier), we collapsed the original method and the copy into a single affected forest, replacing endpoints of semantic edges in either subtree with $*$. Instead, we could distinguish between the original method (collapsing those endpoints to $*_1$) and the copy (collapsing to $*_2$). Then, there would be several more edge classifications, including four different internal edge classifications (internal to $*_1$, internal to $*_2$, edges from $*_1$ to $*_2$, and edges from $*_2$ to $*_1$). While such modifications provide a much finer level of granularity for specifying preservation requirements, they do complicate the language for specifying these requirements—sometimes significantly. It can quickly devolve into (essentially) a language for specifying graph rewriting rules, which have their own subtleties and complications [5]. In our experience, by aggressively decomposing complex refactorings into sequences of smaller refactorings, we have generally found such changes to be unnecessary for the types of refactorings most commonly implemented in refactoring tools.

10 Conclusions & Future Work

In this paper, we classified refactoring preconditions as ensuring input validity, compilability, and behavior preservation, and we proposed a technique for many compilability and preservation preconditions to be checked after transformation in a generic way. We showed that, if essential semantic relationships are treated as edges in a program graph, these edges can be classified based on their relationship to the modified subtree(s). The preservation requirements for common refactorings can be expressed by indicating, for each kind of edge, whether a subset or superset of those edges should be preserved. By exploiting an isomorphism between graph nodes and textual intervals, the preservation checking algorithm can be implemented in a way that is both efficient and language independent. We implemented this technique in a library and applied it to refactorings for Fortran 95, PHP 5, and BC.

Much future work is possible. When differential precondition checking is used, how does it affect the amount of time taken to implement a refactoring? Do refactorings implemented with differential precondition checking tend to have more or fewer bugs than those implemented with traditional precondition checks? Both of these questions will require empirical data from many developers to answer conclusively. What other refactorings can be implemented using the preservation specifications described in this paper? 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 precondition checker with expensive interprocedural analyses for the purposes of testing but to replace these analyses with cheaper, traditional precondition checks in production? We hope that researchers will address these and other questions about differential precondition checking in the future.

Acknowledgment

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing Applications, IBM, and the Great Lakes Consortium for Petascale Computation. Additionally, portions of this research were supported by the National Science Foundation under award CCF-1217271. The authors would like to thank the anonymous reviewers, as well as Rob Bocchino, John Brant, Brett Daniel, Danny Dig, Matthew Fotzler, Milos Gligoric, Vilas Jagannath, Ashley Kasza, Darko Marinov, Stas Negara, and members of the Brett Daniel Software Engineering Seminar for providing invaluable feedback on earlier drafts of this paper.

References

[1] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *FSE '07*.

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

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, January 1995.

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

[5] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *J. Softw. Maint. Evol.*, 17(4):247–276, 2005.

[6] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *ICSE '09*.

[7] Jeffrey L. Overbey, Matthew J. Fotzler, Ashley J. Kasza, and Ralph E. Johnson. A collection of refactoring specifications for Fortran 95, BC, and PHP 5. Technical Report http://jeff.over.bz/papers/2011/tr-refacs.pdf, 2011.

[8] Jeffrey L. Overbey and Ralph E. Johnson. Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In *ASE '11*.

[9] Jeffrey L. Overbey and Ralph E. Johnson. Generating rewritable abstract syntax trees. In *SLE 2008*, volume 5452 of *LNCS*, pages 114–133.

[10] Refactoring benchmarks for extract method. `http://c2.com/cgi/wiki?RefactoringBenchmarksForExtractMethod`.

[11] Refactoring benchmarks for pull up method. `http://c2.com/cgi/wiki?RefactoringBenchmarksForPullUpMethod`.

[12] Christoph Reichenbach, Devin Coughlin, and Amer Diwan. Program metamorphosis. In *ECOOP '09*.

[13] Max Schäfer and Oege de Moor. Specifying and implementing refactorings. In *SPLASH '10*.

[14] Max Schäfer, Julian Dolby, Manu Sridharan, Frank Tip, and Emina Torlak. Correct refactoring of concurrent Java code. In *ECOOP '10*.

[15] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for Java. In *OOPSLA '08*.

[16] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon – lightweight language extensions to easily realise refactorings. In *ECOOP '09*.

[17] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In *ICSE'06*.