Change Impact Analysis for Object-Oriented Programs

Barbara G. Ryder* and Frank Tip
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
ryder@cs.rutgers.edu, tip@watson.ibm.com

ABSTRACT

Small changes can have major and nonlocal effects in object-oriented languages, due to the use of subtyping and dynamic dispatch. This complicates life for maintenance programmers, who need to fix bugs or add enhancements to systems originally written by others. *Change impact analysis* provides feedback on the semantic impact of a set of program changes. This analysis can be used to determine the regression test drivers that are affected by a set of changes. Moreover, if a test fails, a *subset* of changes responsible for the failure can be identified, as well as a subset of changes that can be incorporated safely without affecting any test driver.

1. INTRODUCTION

Object-oriented programming languages present many challenges for program understanding. The extensive use of subtyping and dynamic dispatch make understanding the flow of values and control a nontrivial task. Moreover, small source code changes can have unexpected and nonlocal effects. For example, adding a method to an existing class may affect the dispatch behavior of virtual method calls throughout the program. Addition of a `new` statement can cause a new receiver type to reach a virtual call site and thereby result in a call to a different callee, arbitrarily far from the added `new`. This *nonlocality of change impact* is qualitatively different and more important for object-oriented programs than for imperative programs; for example, in C programs a precise call graph can be derived from syntactic information alone, except for the typically few calls through function pointers. As a result, maintenance programmers, who need to fix bugs or add enhancements to object-oriented systems are often hesitant to make invasive changes because of the unforeseen effects that these changes might have.

This paper is concerned with *change impact analysis*, a collection of techniques for determining the effects of a set of source code changes. In this approach, the first step consists of mapping the source code changes to a set of *atomic* changes. In order to keep our analysis simple and scalable, we use classes, methods, fields and their interrelationships as the atomic units of change. Furthermore, a *partial order* between these atomic changes is determined. Intuitively, this partial order captures dependences between the changes that must be respected so as to create a syntactically valid program. Then, for a given set A of atomic changes, and a given set T of test drivers that exercise parts of the program's functionality, a static analysis is performed to determine:

- A subset T' of the test drivers in T that are potentially affected by changes in A. This information can be used for regression test selection [10].
- A subset A' of the changes in A that may affect a specific test driver t in T. This allows programmers to ignore any change that is not involved in t's failure. Moreover, we introduce a notion of *dependence* among atomic changes that enables one to construct compilable programs that incorporate some, but not all the changes in A'.
- A subset of changes in A that do not affect any test in T. These changes can be incorporated immediately, without breaking any test.
- Coverage information that informs the programmer about code not yet covered by tests that can serve as a basis for creating new tests.

We use call graphs as the basis for the above analysis. Recent work on call graph construction algorithms by one of the authors [13] has led us to believe that call graphs can be computed precisely and efficiently enough to support the above analyses in an interactive tool setting.

The long-term goal of our project is to incorporate change impact analysis into an existing IDE such as IBM's VisualAge Java[1]. This will be part of a larger effort to provide analysis-based support for refactoring [4], program understanding, and regression testing. This project is currently in the design stage, and the present paper focuses primarily on algorithmic and architectural aspects.

2. MOTIVATING EXAMPLE

The Java classes in Figure 1(a) will be used as a running example to illustrate our notion of change impact analysis. The example consists of five classes:

*On sabbatical at IBM T.J. Watson 2000–2001 from Rutgers University.

[1]See `www.ibm.com/software/ad/vajava`.

PASTE'01, June 18-19, 2001, Snowbird, Utah, USA.

```
class Person {
   private String name;
   Person(String n){ name = n; }
   public String getName() { return name;}
   public String toString(){ return name; }
}
class Student extends Person {
   private Set courses;
   Student(String name){ super(name); courses = new HashSet(); }
   public void addCourse(Course c){ courses.add(c); }
   public int totalCredits(){
      int sum = 0;
      for (Iterator en=courses.iterator(); en.hasNext(); ){
         Course c = (Course)en.next(); sum += c.getCredits();
      }
      return sum;
   }
}
class Professor extends Person {
   private String department, office;
   private Set teaching;
   Professor(String name, String d, String off){
      super(name); department = d;
      office = off; teaching = new HashSet();
   }
   public void addCourse(Course c){ teaching.add(c); }
   public int load(){ return teaching.size(); }
   public String toString(){
      return (super.toString() + ", office at " +
      office + " department is " + department);
   }
}
class Course {
   private String id; private int credits;
   private Set students; private Professor p;
   Course(String n; Professor pp; int c){
      id = n; p = pp; credits = c; students = new HashSet();
   }
   public void addStudent(Student x){students.add(x);}
   public String getId(){ return id; }
   public HashSet getStudents{ return students; }
   public Professor getProfessor(){ return p; }
   public void setProfessor(Professor pp){ p = pp; }
   public int getCredits(){ return credits; }
}
class University {
   private Set people;
   University(){ people = new HashSet(); }
   public Set getPeople(){ return people; }
   public void addPerson(Person p){ people.add(p);}
   public Professor findProfessor(String name){
      for (Iterator en = people.iterator(); en.hasNext(); ){
         Person p = (Person)en.next();
         if (p instanceof Professor && name.equals(p.getName()))
            return (Professor)p;
      }
      return null;
   }
   public Student findStudent(String name){
      for (Iterator en = people.iterator(); en.hasNext(); ){
         Person p = (Person)en.next();
         if (p instanceof Student && name.equals(p.getName()))
            return (Student)p;
      }
      return null;
   }
   public void assignProf(Professor p, Course c){
      p.addCourse(c); c.setProfessor(p);
   }
   public void enrollinCourse(Student s, Course c){
      s.addCourse(c); c.addStudent(s);
   }
}
```

(a)

```
class TestA {
   public static void main(String args[]){
      University u = new University();
      Professor p1 = new Professor("Barbara Ryder","DCS","CORE 311");
      u.addPerson(p1);
      Course c1 = new Course("100", p1,4);
      University.assignProf(p1,c1);
      p1 = new Professor("Frank Tip", "DCS", "CORE 320");
      u.addPerson(p1);
      Course c2 = new Course("200",p1,3);
      u.assignProf(p1,c2);
      Professor q = u.findProfessor("Donald Smith");
      if (q != null){
         System.out.println("Professor Donald Smith found");
         System.out.println(q.toString() +
            " is teaching " + q.load() + " courses.");
      } else {
         System.out.println("Professor Donald Smith not found");
      }
      Professor p = u.findProfessor("Barbara Ryder");
      if (p != null){
         System.out.println("Professor Barbara Ryder found");
         System.out.println(p.toString() +
            " is teaching " + p.load() + " courses.");
      } else {
         System.out.println("Professor Barbara Ryder not found");
      }
   }
}
class TestB {
   public static void main(String args[]){
      University u = new University();
      u.addPerson(new Professor("Barbara Ryder","DCS","CORE 311"));
      u.addPerson(new Professor("Frank Tip", "DCS", "CORE 320"));
      u.addPerson(new Student("Atanas Rountev"));
      u.addPerson(new Student("Matt Arnold"));
      String s = "";
      for (Iterator en = u.getPeople().iterator(); en.hasNext(); ){
         Person p = (Person)en.next(); s += p.toString();
      }
      System.out.println("University people are " + s);
   }
}
class TestC {
   public static void main(String args[]){
      University u = new University();
      Student s1 = new Student("Atanas Rountev");
      Student s2 = new Student("Matt Arnold");
      u.addPerson(s1); u.addPerson(s2);
      Course c1 = new Course("100",null,4);
      Course c2 = new Course("200",null,3);
      u.enrollinCourse(s1,c1);
      u.enrollinCourse(s1,c2);
      u.enrollinCourse(s2,c1);
      Student s3 = u.findStudent("Matt Arnold");
      Student s4 = u.findStudent("Ana Milanova");
      if (s3 != null){
         System.out.println(s3.toString() + " is taking"+
            s3.totalCredits() + " credits");
      } else {
         System.out.println("Matt Arnold is not a student");
      }
      if (s4 != null){
         System.out.println(s4.toString() + " is taking " +
            s4.totalCredits() + " credits");
      } else {
         System.out.println("Ana Milanova is not a student");
      }
   }
}
```

(b)

Figure 1: University example. (a) Classes `Person`, `Student` `Professor`, `Course`, and `University`. (b) Test drivers `TestA`, `TestB`, and `TestC`.

Course, Person, Professor, Student and University. A University is populated with Persons with the appropriate attributes (e.g., offices, departments). Professors are assigned courses to teach by way of a method University.assignProf() and students are enrolled in courses using a method University.enrollinCourse(). Two methods University.findProfessor() and University.findStudent() are provided to search for professors and students by their name.

Figure 1(b) shows three test driver classes TestA, TestB, and TestC. TestA tests the functionality for finding a particular professor and printing his or her course load. TestB tests the ability to print out all Persons currently at the University. TestC finds a particular student and prints his or her credit load. Each of the test driver classes together with the five university classes form a coherent Java program.

Since we are studying the impact of changes, we need to posit some modifications to the original five class system. The first change is caused by the university adopting an identification number for its students that should always be presented along with the other information associated with a student. This requires the addition of a field Student.idNum to class Student to contain the ID number, a change to the constructor of Student to initialize this field, and the addition of a method Student.toString() to print the student number. Note that some changes are needed in test drivers TestB and TestC in order to create Student objects properly.

Considering the impact of the first change, note that the calls to toString() in TestB and TestC will dispatch to a new method for objects of type Student. Clearly, these tests must be rerun to determine if the altered behavior matches the programmer's expectations. Note how this simple change illustrates the nonlocality of change impact in object-oriented programs: neither TestB nor TestC has any relation to Student in the class hierarchy, and the affected calls to toString() are arbitrarily far away from other methods of Student in the call graph!

The second change occurs due to a new university policy that allows for the association of any person with a department (as opposed to only professors). This involves: (i) adding field department to class Person and removing it from class Professor, (ii) adding a second constructor to class Person that initializes the department as well as the name, (iii) changing Professor's constructor (removing the assignment to Professor.department, and passing d as an extra argument in the super-call), (iv) changing Person.toString() to print out the name and department if the latter is available, and otherwise only the name, and (v) changing Professor.toString() (removing the printing of Professor.department). Due to the change in Person.toString(), all test drivers now execute changed code; however, the output produced by each test case is the same as before.

Finally, a third system change occurs when the university caps course enrollment at a maximum of 50 students. This is implemented by inserting an **if**-statement in University.enrollinCourse(). Only TestC, which calls this method, is affected.

3. CHANGES

Our analysis assumes the existence of an original program P and a changed program P' derived from P. Both P and P' are assumed to be syntactically correct and compilable, but we impose no restrictions on the number or the nature of the changes that transform P into P'. We assume that an IDE provides information about the files, classes, and methods that have been edited. Alternatively, one can rely on a utility like *diff* to obtain this information.

(**AC**)	Add an empty class
(**DC**)	Delete an empty class
(**AM**)	Add an empty method
(**DM**)	Delete an empty method
(**CM**)	Change body of method
(**LC**)	Change virtual method lookup
(**AF**)	Add a field
(**DF**)	Delete a field

Table 1: Categories of atomic changes.

3.1 Atomic Changes

A key aspect of our approach is the ability to transform source code edits into a set of *atomic changes*, as defined in Table 1. These have two important characteristics. First, their granularity matches our analysis; that is, our analysis will not be able to produce more precise results if a finer-grained (e.g., statement-oriented) notion of atomic change is used. Second, any source code edit can be broken up into a *unique* set of atomic changes. Most of the changes in Table 1 are self-explanatory, except for **CM** and **LC**. **CM** captures any kind of change to a method body, including (i) adding a body to a previously abstract method, (ii) removing the body of a non-abstract method and making it abstract, and (iii) making any number of statement-level changes inside a method body. The **LC** category "abstracts" any kind of source code change that affects dynamic dispatch behavior[2].

LC changes can be caused by adding or deleting methods, and by adding or deleting inheritance relations. The computation of **LC** changes is somewhat involved, and will be discussed below in Section 3.2.

For a given source code edit, we will use the labels of Table 1 to denote the sets of atomic changes derived from that edit. In other words, **AM**, **CM**, and **DM** denote sets of added, changed, and deleted methods, respectively. Similarly, **AF** and **DF** denote added and deleted fields, and **AC** and **DC** denote sets of added and deleted classes, respectively.

We will ignore several kinds of source code level changes that have no direct semantic impact apart from controlling visibility and thereby compilability. These include changes to access rights of classes, methods, and fields, addition/deletion of comments, and addition/deletion of import statements.

3.2 Changes affecting method dispatch

As mentioned, method dispatch behavior may be affected by several kinds of edits. Before we can reason about *changes* in method dispatch behavior, we formalize the method dispatch process using a function *Lookup*. *Lookup* takes two ar-

[2]Some source changes correspond to more than one atomic change. For example, the addition of an empty method may imply several atomic changes, of types **AM** and **LC**. Here, the **AM** change denotes the added method as a node in the call graph of P', and the **LC** change(s) specifies the change(s) in dynamic dispatch behavior caused by this method addition.

$$
\begin{aligned}
\textit{Lookup} \quad &= \{ \langle C, A.m, B.m \rangle \mid \text{class } A \text{ contains virtual method } m,\ C \leq^* B \leq^* A, \\
&\qquad \text{class } B \text{ contains virtual method } m, \\
&\qquad \text{there is no class } B' \text{ that contains method } m \text{ such that } C \leq^* B' <^* B \} \\
\textbf{LC} \quad &= \{ \langle a, b \rangle \mid \langle a, b, c \rangle \in ((\textit{Lookup}_{old} - \textit{Lookup}_{new}) \cup (\textit{Lookup}_{new} - \textit{Lookup}_{old})) \} \\
\textit{SameLookup}(B, f()) \quad &= \{ \langle C, Y.f(), V.f() \rangle \mid \langle B, Y.f(), V.f() \rangle \in \textit{Lookup}_{old},\ \langle C, Y.f(), V.f() \rangle \in \textit{Lookup}_{old},\ C \leq^* B \leq^* Y \}
\end{aligned}
$$

Figure 2: Definitions of *Lookup*, LC, and *SameLookup*.

guments, the run-time type of the receiver and the method that is statically referred to in the method call, and returns the method definition that is invoked by the virtual dispatch mechanism[3]. If we consider the lookup function to be a map, that is a set of triples ⟨ *runtimeReceiverType*, *staticMethodSignature*, *actualMethodBound* ⟩, then some of the values corresponding to our original hierarchy of Figure 1 are:

⟨ `Person`, `Person.toString()`, `Person.toString()` ⟩
⟨ `Student`, `Person.toString()`, `Person.toString()` ⟩
⟨ `Professor`, `Person.toString()`, `Professor.toString()` ⟩
⟨ `Professor`, `Professor.toString()`, `Professor.toString()` ⟩
⟨ `Person`, `Person.getName()`, `Person.getName()` ⟩
⟨ `Professor`, `Person.getName()`, `Person.getName()` ⟩
⟨ `Student`, `Person.getName()`, `Person.getName()` ⟩

Using the inheritance operators of Table 2, the complete *Lookup* map can be defined as the set of all tuples ⟨ C, $A.m$, $B.m$ ⟩ such that $A.m$ is a method declared in the hierarchy, $C \leq^* A$, and B is nearest superclass of C containing a definition of method m. Figure 2 shows the definition of *Lookup*.

Having defined *Lookup*, we now turn our attention to *changes* in lookup behavior. For convenience, we will use $\textit{Lookup}_{old}$ and $\textit{Lookup}_{new}$ to refer to the set of *Lookup* tuples before and after the edit, respectively. We can now define **LC** as a set of pairs ⟨ C, $A.m()$ ⟩, indicating that the dynamic dispatch behavior for a call to $A.m()$ on an object with run-time type C has changed. The definition of **LC** is also given in Figure 2.

It is possible to compute **LC** by directly following the definition, and traversing the class hierarchy for each run-time type and each method signature. However, re-traversing the *entire* hierarchy after each edit seems unnecessarily expensive, since it is likely that large parts of the *Lookup* map are unaffected by an edit. We therefore plan to pursue an approach where the *Lookup* map is *updated* after each edit instead of being recomputed from scratch. In the remainder of this section, we will study a number of typical edits, the corresponding changes to *Lookup*, and the effect on **LC**. The edits usually require removing some invalidated tuples from *Lookup*, and adding some newly created tuples. For example, when a method $B.f()$ is added, we may remove some tuples from $\textit{Lookup}_{old}$ that resolve references with run-time types corresponding to the inheritance tree rooted at B to actual methods defined in ancestors of B. Then, new tuples are added that express how some references with run-time type B (or a subtype of B) are resolved in the updated hierarchy to the newly added $B.f()$. Table 3 contains a summary of several edits and their impact on **LC**. In other words, the table states how, for several edits, $\textit{Lookup}_{new}$ can be computed directly from $\textit{Lookup}_{old}$ without referring to the hierarchy. The remainder of this section will give examples of the edits shown in Table 3.

We conclude this discussion by introducing an auxiliary function *SameLookup*, which will be used to define the impact of several of the edits in Figure 3. *SameLookup* takes two arguments, a run-time type B, and a method signature $f()$, and computes the subset of $\textit{Lookup}_{old}$ comprising all tuples ⟨ C, $Y.f()$, $V.f()$ ⟩ $\in \textit{Lookup}_{old}$ for which there is a corresponding element ⟨ B, $Y.f()$, $V.f()$ ⟩ $\in \textit{Lookup}_{old}$. Informally, the purpose of *SameLookup* is to identify all calls that are resolved the same as a call to a method $f()$ on a B-object. The definition of *SameLookup* can be found in Figure 2.

[3]This only applies to dynamically dispatched (virtual) methods. Hence, *Lookup* and **LC** do not contain tuples for constructors and static methods.

$A < B$	A is a direct descendent of B
$A \leq B$	A is a direct descendent of B, or $A = B$
$A \leq^* B$	B is an ancestor of A, or $A = B$
$A <^* B$	B is an ancestor of A, but $B \neq A$

Table 2: Notation for inheritance relations.

3.2.1 Edit I: method addition

In this example, we study the impact of the addition of a method `getName()` to class `Professor` of Figure 1. This change will result in the deletion of element:

⟨ `Professor`, `Person.getName()`, `Person.getName()` ⟩

from $\textit{Lookup}_{old}$, because invoking method `Person.getName()` on an object of type `Professor` will no longer resolve to `Person.getName()`. Observe that new elements have to be added as well: invoking `Person.getName()` on a `Professor`-object will resolve to `Professor.getName()`, and invoking `Professor.getName()` on a `Professor`-object will resolve to `Professor.getName()` as well. Hence, the following tuples need to be added:

⟨ `Professor`, `Person.getName()`, `Professor.getName()` ⟩
⟨ `Professor`, `Professor.getName()`, `Professor.getName()` ⟩

Consequently, for this edit we have that:

$$\textbf{LC} = \{ \ \langle \texttt{Professor}, \texttt{Person.getName()} \rangle, \ \langle \texttt{Professor}, \texttt{Professor.getName()} \rangle \ \}$$

Table 3 states how method addition affects *Lookup* in the presence of overriding definitions of the added method in subclasses.

3.2.2 Edit II: method deletion

We will now consider the impact of deleting method `Professor.toString()` on *Lookup*. First, we need to re-

edit	deleted tuples	added tuples
add method $B.f()$	$\mathit{SameLookup}(B, f())$	$\{ \langle C, X.f(), B.f() \rangle \mid \langle C, X.f(), Y.f() \rangle \in \mathit{SameLookup}(B, f()), C \leq^* B <^* Y \leq^* X \}$ $\{ \langle C, B.f(), B.f() \rangle \mid f$ is not defined in any B', $C \leq^* B' \leq^* B \}$ $\{ \langle E, B.f, D.f \rangle \mid \langle E, X.f(), D.f() \rangle \in \mathit{Lookup}_{old}, E \leq^* D <^* B \}$
delete method $B.f()$	$\{ \langle C, A.f, B.f \rangle \mid C \leq^* B \leq^* A \}$ $\{ \langle D, B.f, C.f \rangle \mid D \leq^* C \leq^* B \}$	$\{ \langle C, X.f(), A.f() \rangle \mid \langle C, X.f(), B.f() \rangle \in \mathit{Lookup}_{old}, X \neq B$ A is B's closest ancestor that defines $f()$, $C \leq^* B \leq^* A \}$
add leaf class C	*none*	$\{ \langle C, X.f(), V.f() \rangle \mid \langle B, X.f(), V.f() \rangle \in \mathit{SameLookup}(B, f()), C < B \}$
delete leaf class C	$\{ \langle C, X.f(), V.f() \rangle \mid C \leq^* X \leq^* V \}$	*none*
move nonempty leaf class B ($B < A$ to $B < D$)	$\{ \langle B, X.f(), Y.f() \rangle \mid B < A \leq^* X,$ $\langle B, X.f(), Y.f() \rangle \in \mathit{Lookup}_{old} \}$	$\{ \langle B, X.f(), V.f() \rangle \mid \langle D, X.f(), V.f() \rangle \in \mathit{SameLookup}(D, f()),$ $f()$ not defined in class $B \}$ $\{ \langle B, X.f(), B.f() \rangle \mid \langle D, X.f(), V.f() \rangle \in \mathit{SameLookup}(D, f()),$ $f()$ is defined in class $B \}$
move subtree rooted at B ($B < A$ to $B < D$)	$\{ \langle C, X.f(), Y.f() \rangle \mid C \leq^* B < A \leq^* X$ $\langle C, X.f(), Y.f() \rangle \in \mathit{Lookup}_{old} \}$	$\{ \langle C, X.f(), Q.f() \rangle \mid \langle C, R.f(), Q.f() \rangle \in \mathit{Lookup}_{old}, C \leq^* Q \leq^* R \leq^* B,$ $\langle D, X.f(), Y.f() \rangle \in \mathit{SameLookup}(D, f()), D \leq^* Y \leq^* X \}$ $\{ \langle C, X.f(), Y.f() \rangle \mid \nexists \langle C, R.f(), Q.f() \rangle \in \mathit{Lookup}_{old}$ for $C \leq^* Q \leq^* R \leq^* B,$ $\langle D, X.f(), Y.f() \rangle \in \mathit{SameLookup}(D, f()), D \leq^* Y \leq^* X \}$

Table 3: An overview of several typical edit actions, and their impact on *Lookup*. From left to right, the columns state: a description of the edit, the set of tuples removed from *Lookup* as a result of the edit, and the set of tuples added to *Lookup* as a result of the edit, respectively. The method f in the last four rows of the table is meant to vary over all methods that exist in the hierarchy before the edit.

move the elements of *Lookup* associated with invoking a `toString()` method on a `Professor`-object:

$\langle$ `Professor`, `Person.toString()`, `Professor.toString()` $\rangle$
$\langle$ `Professor`, `Professor.toString()`, `Professor.toString()` $\rangle$

Then, we must add elements to *Lookup* that reflect the fact that invoking `toString()` on a `Professor`-object now resolves to `Person.toString()`:

$\langle$ `Professor`, `Person.toString()`, `Person.toString()` $\rangle$

Hence, we have that:

LC = { $\langle$`Professor,Person.toString()`$\rangle$,
$\langle$`Professor,Professor.toString()`$\rangle$ }

Table 3 states how method deletion affects *Lookup* in the more general case, where the deleted method is overridden in subclasses.

3.2.3 Edit III: addition of an empty leaf class

If we add classes `GradStud` and `UgStud` to Figure 1, then we will have `GradStud` < `Student` and `GradStud` <* `Person` in the resulting hierarchy. In this case, new tuples will need to be added to *Lookup* to reflect the resolution of methods that are defined in classes `Person` and `Student` on objects of type `GradStud` and `UgStud`:

$\langle$ `GradStud`, `Person.toString()`, `Person.toString()` $\rangle$
$\langle$ `GradStud`, `Person.getName()`, `Person.getName()` $\rangle$
$\langle$ `GradStud`, `Student.addCourse()`, `Student.addCourse()` $\rangle$
$\langle$ `GradStud`, `Student.totalCredits()`, `Student.totalCredits()` $\rangle$
$\langle$ `UgStud`, `Person.toString()`, `Person.toString()` $\rangle$
$\langle$ `UgStud`, `Person.getName()`, `Person.getName()` $\rangle$
$\langle$ `UgStud`, `Student.addCourse()`, `Student.addCourse()` $\rangle$
$\langle$ `UgStud`, `Student.totalCredits()`, `Student.totalCredits()` $\rangle$

Hence, the impact on **LC** is as follows:

LC = { $\langle$`GradStud,Person.toString()`$\rangle$,
$\langle$`GradStud,Person.getName()`$\rangle$,
$\langle$`GradStud,Student.addCourse()`$\rangle$,
$\langle$`GradStud,Student.totalCredits()`$\rangle$,
$\langle$`UgStud,Person.toString()`$\rangle$,
$\langle$`UgStud,Person.getName()`$\rangle$,
$\langle$`UgStud,Student.addCourse()`$\rangle$,
$\langle$`UgStud,Student.totalCredits()`$\rangle$ }

3.2.4 Edit IV: deletion of an empty leaf class

We will use the example program of Figure 1 augmented with the additional classes `GradStud` and `UgStud` of Section *3.2.3* as the basis for the next example. Here, we consider the deletion of class `UgStud`. This implies that all tuples whose run-time type component is `UgStud` will be removed from *Lookup*, i.e.:

$\langle$ `UgStud`, `Person.toString()`, `Person.toString()` $\rangle$
$\langle$ `UgStud`, `Person.getName()`, `Person.getName()` $\rangle$
$\langle$ `UgStud`, `Student.addCourse()`, `Student.addCourse()` $\rangle$
$\langle$ `UgStud`, `Student.totalCredits()`, `Student.totalCredits()` $\rangle$

Hence, the impact on **LC** is:

LC = { $\langle$`UgStud,Person.toString()`$\rangle$,
$\langle$`UgStud,Person.getName()`$\rangle$,
$\langle$`UgStud,Student.addCourse()`$\rangle$,
$\langle$`UgStud,Student.totalCredits()`$\rangle$ }

3.2.5 Edit V: move a class

We will now consider moving a class in the hierarchy using the example program of Figure 1. In order to accommodate adjuncts in our University model, we will create a new class `Teacher` as a child of `Person`, add `Adjunct` as its child, and then move `Professor` to be its second child. We can add `Teacher` and `Adjunct` directly using the transformations of Section *3.2.3*. `Teacher` will be added as an empty leaf class and therefore, the changes to *Lookup* will consist of adding tuples to represent those functions inherited from `Person`:

$\langle$ `Teacher`, `Person.getName()`, `Person.getName()` $\rangle$
$\langle$ `Teacher`, `Person.toString()`, `Person.toString()` $\rangle$, etc.

Similar updates will be necessary to add `Adjunct`. However, to move `Professor` to become a child of `Teacher`, we cannot use this transformation because it assumes the class to be empty, and the existing `Professor` class is not. In general, when we move a leaf class B from being a child of A to being a child of D, we must remove all tuples corresponding to methods inherited through A and add all newly inherited methods from D, the new parent of B. When B is the root of a subtree in the inheritance hierarchy, we must be careful not to lose existing overrides of functions within the subtree rooted at B. Before moving the `Professor` class, $Lookup_{old}$ contains the following tuple due to inheritance from `Person`:

$\langle$ `Professor`, `Person.getName()`, `Person.getName()` $\rangle$

When `Professor` becomes a child of `Teacher`, we must remove this tuple. Then, we need to add tuples to $Lookup_{new}$ that correspond to methods inherited through `Teacher`. Assume we have added some methods to `Teacher` before moving `Professor` to be its child. If a method `Teacher.numCourses()` exists, then the set of added tuples includes:

$\langle$ `Professor`, `Teacher.numCourses()`, `Teacher.numCourses()` $\rangle$
$\langle$ `Professor`, `Person.getName()`, `Person.getName()` $\rangle$, etc.

Notice that we have deleted and then added the following tuple in this update:

$\langle$ `Professor`, `Person.getName()`, `Person.getName()` $\rangle$

This happens because `Professor`'s old and new parents share a common ancestor `Person`, from which both inherit `Person.getName()`. In addition, notice that we have not added or deleted:

$\langle$ `Professor`, `Professor.toString()`, `Professor.toString()` $\rangle$

because method calls that involve a method inside class `Professor` on an object with run-time type `Professor` are not affected by the move of class `Professor` to its new location in the hierarchy.

The example that we just discussed corresponds to the edit *move nonempty leaf class B* in Table 3. The edit *move subtree rooted at B* in Table 3 is handled similarly, but updates must account in addition for changes in dynamic dispatch for subclasses of B.

3.3 Ordering atomic changes

Changes may depend on other changes, both syntactically and semantically. For the purposes of this paper, we will only consider syntactic dependences that must be satisfied to ensure compilability. Examples of syntactic dependences are that one cannot extend a class that does not exist, or call a method that has not been defined yet. An example of a semantic dependence is where a new method m only exhibits correct behavior in the presence of a changed version of a method m' that it calls. Section 5 will present several scenarios in which a change impact analysis tool that is aware of dependences between changes can provide valuable support to users when a test case fails after a set of changes is applied. This ability to explore partial edits of the program is quite useful.

We express syntactic dependence between changes using a partial ordering $\prec$ on atomic changes (with transitive closure $\preceq^*$). For a given set A of atomic changes that transforms P into P', $\prec$ can be used to determine *consistent* subsets A' of A such that applying A' to P results in a valid (i.e., compilable) program P'' that incorporates some, but not all of the changes in P'. A subset A' of the full set of atomic changes A is *consistent* if:

$$\forall a' \in A \text{ such that } a' \preceq^* a, a \in A' \Rightarrow a' \in A'$$

We plan to compute the ordering between atomic changes automatically, without intervention by the programmer. Computation of this ordering requires determining the "syntactic requirements" of program fragments referenced in an atomic change. While we do not expect this to be difficult, a detailed formalization is future work.

3.4 Deriving atomic changes

Breaking up source code edits into atomic changes is fairly straightforward. Due to space limitations we only demonstrate this process by example.

With respect to our example in Figure 1, the first edit described in Section 2 was the addition of a student ID number to the program. This edit corresponds to the following atomic changes: $c_1 \equiv$ `Student.idNum` $\in$ **AF**, $c_2 \equiv$ `Student.Student()` $\in$ **CM**, $c_3 \equiv$ `Student.toString()` $\in$ **AM**, $c_4 \equiv$ `Student.toString()` $\in$ **CM**, $c_5 \equiv \langle$ `Student`, `Student.toString()` $\rangle \in$ **LC**, and $c_6 \equiv \langle$ `Student`, `Person.toString()` $\rangle \in$ **LC**. Here, we have that $c_1 \prec c_2$, $c_1 \prec c_3 \prec c_4$, $c_3 \prec c_5$, and $c_3 \prec c_6$.

The second edit allowed each person to be affiliated with a department. This edit corresponds to the following atomic changes: $c_7 \equiv$ `Person.department` $\in$ **AF**, $c_8 \equiv$ `Professor.department` $\in$ **DF**, $c_9 \equiv$ `Person.Person(String,String)` $\in$ **AM**, $c_{10} \equiv$ `Person.Person(String,String)` $\in$ **CM**, $c_{11} \equiv$ `Professor.Professor()` $\in$ **CM**, $c_{12} \equiv$ `Person.toString()` $\in$ **CM**, and $c_{13} \equiv$ `Professor.toString()` $\in$ **CM**. These changes are ordered as follows: $c_7 \prec c_{10}$, $c_7 \prec c_{12}$, $c_9 \prec c_{10}$, and $c_9 \prec c_{11}$.

The third edit implements the new rule that caps course enrollment at 50 students. This corresponds to one atomic change, $c_{14} \equiv$ `University.enrollInCourse()` $\in$ **CM**.

4. CHANGE IMPACT ANALYSIS

We will assume that associated with program P is a set of test drivers $\mathcal{T} = t_1, \cdots, t_n$. Each test driver t_i exercises a subset $Nodes(P, t_i)$ of P's methods, and a subset $Edges(P, t_i)$ of calling relationships between P's methods. Likewise, $Nodes(P', t_i)$ and $Edges(P', t_i)$ form the call graph for t_i on the edited program P'. Here, a calling relationship between methods is assumed to be of the form $A.m \rightarrow_C B.n$, indicating that control may flow from method $A.m$ to method $B.n$ due to a virtual call to method n on an object of type C.

We do not require full coverage (i.e., that every method in P be exercised by at least one test driver), nor that test drivers exercise disjoint fragments of code. However, our analyses are likely to be most effective in situations where many test drivers each exercise a small part of a system's functionality, under approximately the above conditions.

$$AffectedTests(\mathcal{T}, \mathcal{A}) = \{ t_i \mid t_i \in \mathcal{T},\ Nodes(P, t_i) \cap (\mathbf{CM} \cup \mathbf{DM})) \neq \emptyset \} \cup \{ t_i \mid t_i \in \mathcal{T},\ n \in Nodes(P, t_i),\ n \rightarrow_B A.m \in Edges(P, t_i),\ \langle B, X.m \rangle \in \mathbf{LC},\ B <^* A \leq^* X \}$$

$$AffectingChanges(t, \mathcal{A}) = \{ a' \mid a \in Nodes(P', t) \cap (\mathbf{CM} \cup \mathbf{AM}),\ a' \preceq^* a \} \cup \{ a' \mid a \equiv \langle B, X.m \rangle \in \mathbf{LC},\ B <^* A \leq^* X,\ n \rightarrow_B A.m \in Edges(P', t), \text{ for some } n, A.m \in Nodes(P', t),\ a' \preceq^* a \}$$

Figure 3: Change impact analysis definitions.

Figure 3 shows definitions of the two key concepts that form the foundation of our analysis. $AffectedTests(\mathcal{T},\mathcal{A})$ is a subset of $\mathcal{T}$ containing only those test drivers whose behavior may be affected by changes in $\mathcal{A}$. This comprises any test driver that traverses a changed or deleted method, as well as any test driver that contains a virtual dispatch whose behavior may have changed. $AffectingChanges(t,\mathcal{A})$ is a subset of the changes in $\mathcal{A}$ that may affect the behavior of a specific test driver t. Observe that these definitions do not rely on any particular method for determining *Nodes* and *Edges*[4]. We plan to experiment with efficient call graph construction algorithms such as RTA [1] and XTA [13], but using trace information gathered at run-time is another possibility.

AffectedTests and *AffectingChanges* can be exploited for regression test selection and fault localization as follows:

- Any test driver not in $AffectedTests(\mathcal{T},\mathcal{A})$ is guaranteed to produce the same result after incorporating the changes in $\mathcal{T}$. Hence, only test cases in *AffectedTests* need to be re-executed and have their results examined by the programmer.
- *AffectingChanges* can be used to identify a subset of the changes that do not affect any driver and that can be incorporated safely. However, such changes may be indicative of missing test cases, of which programmers should be made aware.
- *AffectingChanges* can provide useful information once a test driver has failed, by allowing the programmer to focus on failure-related changes.

Let $T = \{ \texttt{TestA}, \texttt{TestB}, \texttt{TestC} \}$. Returning to the first edit of our running example, we can see that atomic change c_2 causes the inclusion of `TestB` and `TestC` in $AffectedTests(T, \{ c_1, c_2, c_3, c_4, c_5, c_6 \})$, because the method changed by c_2 (the constructor of class `Student`) occurs in $Nodes(P, \texttt{TestB})$ and in $Nodes(P, \texttt{TestC})$. However, we find that `TestA` is not affected by the first edit.

Moreover, consider the situation *after all three edits have been applied*, and suppose we are interested in determining which of the atomic changes impacted `TestA` because its behavior is not as expected. To answer this question, we determine $AffectingChanges(\texttt{TestA}, \{ c_1, \ldots, c_{14} \}) = \{ c_7, c_9, c_{10}, c_{11}, c_{12}, c_{13} \}$. In other words, our techniques can detect automatically that neither the first edit (adding the student ID number) nor the third edit (limiting course enrollment) affects `TestA`.

5. TOOL SUPPORT

We plan to implement the concepts of Section 4 as a tool in an IDE. Assume the user edits a program P, makes several changes and then hits a button labeled "analyze change impact". Our tool will determine the set of potentially affected test drivers using *AffectedTests*, and for each driver, the corresponding *AffectingChanges* and its consistent subsets.

Scenario 1. If the programmer makes an edit that adds functionality to the program and the set *AffectedTests* is empty, (i.e., our tool finds no impact), then none of the test drivers are affected by the edit. This might occur when new, non-overriding methods are added, requiring new test drivers. By displaying the edit in terms of its constituent atomic changes, the tool will help to identify new calls and object creations needed for testing the new code.

Scenario 2. Alternatively, our tool may find a nonempty *AffectedTests* set. In this case, the programmer may need to modify an affected test driver, (e.g., change a method signature) in order for it to compile with the edited program. By displaying the *AffectingChanges* set, our tool can show method signature modifications (e.g., added/deleted parameters) that need to be taken into account.

Scenario 3. After all test drivers compile, an affected test driver can produce incorrect results. Assume the set of consistent subsets of *AffectingChanges* corresponding to this driver is A_t. Two possible strategies can be followed to localize the fault. In the first strategy, the tool creates a linear ordering of A_t, and elements of A_t are applied to P in order until the fault is exposed. In the second strategy, binary search is used on A_t to find the smallest set of consistent subsets that still exhibits the fault (similar to [15]). At each step we continue with those changes that expose the fault. Eventually, we reach a smallest set of fault-demonstrating changes.

6. RELATED WORK

Zeller [15] introduced the *delta debugging* approach for localizing failure-inducing changes among large sets of textual changes. His approach involves partitioning changes into subsets, executing the programs resulting from applying these subsets, and determining whether the result is correct, incorrect, or inconclusive. Efficient binary-search-like techniques are used to quickly narrow down the search space. The key differences with our work are that our atomic changes and interdependences take into account program syntax to ensure compilability. Zeller aims at scenarios where new versions of software are supplied by a third party, whereas we are interested in interactive settings where programmers make changes.

Change impact analysis is related both to program slicing [12] and to incremental data-flow analysis [7]. Kung *et al.* have described various sorts of relationships between

[4]In these formulae, assume that **AM**, **DM**, **CM** are encoded as nodes in the call graph of P or P'.

classes and other entities in C++ programs, and presented a technique for determining change impact through these relations [6].

Regression testing validates systems that evolve over time by rerunning tests after every major edit to ensure that functionality has been preserved. *TestTube* [3] and *DejaVu* [10] were designed to diminish the cost of regression testing C programs through analysis, and have recently been compared empirically [2]. We are also interested in determining affected test drivers, but we rely on method-level coverage as opposed to module-level (*TestTube*) or statement-level (*DejaVu*) coverage. Our primary interest is in assisting maintenance programmers with understanding the impact of their program edits, whereas the *TestTube* and *DejaVu* projects emphasize cost reduction for regression testing.

There has been relevant work in adapting procedural testing technology to object-oriented languages. Perry and Kaiser [8] adapted Weyuker's test adequacy rules for procedural languages [14] to account for consequences of virtual dispatch and subtyping. Initial work on data-flow testing of object-oriented programs includes [5, 11]. Other work has suggested selective regression testing for a class-based test methodology [9].

7. FUTURE WORK

Future work at the conceptual level includes a formalization of (i) deriving a set of atomic changes from a source code edit, and (ii) computation of the ordering between atomic changes. We intend to implement the techniques presented in this paper, and assess their effectiveness in practice. We also plan to investigate non-syntactic notions of dependence among atomic changes, in order to reduce the number of partially edited programs that a user needs to consider when faced with a test failure.

In implementing these ideas in a refactoring/change impact tool, we will explore how to best engineer the methodology presented for ease of use and efficient performance. Of interest are actual change histories of existing object-oriented systems, which can be examined to discern patterns of edits (i.e., changes and refactorings) that are common.

Acknowledgments

We are grateful to the anonymous PASTE referees for their comments.

8. REFERENCES

[1] BACON, D. F. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, University of California, Berkeley, Dec. 1997.

[2] BIBLE, J., ROTHERMEL, G., AND ROSENBLUM, D. A comparative study of coarse- and fine-grained safe regression test selection techniques. *ACM Trans. on Software Engineering Methodology* (in press).

[3] CHEN, Y., ROSENBLUM, D., AND VO, K. Testtube: A system for selective regression testing. In *Proc. of the 16th Int. Conf. on Software Engineering* (1994), pp. 211–220.

[4] FOWLER, M. *Refactoring.* Addison-Wesley, 1999.

[5] HARROLD, M. J., AND ROTHERMEL, G. Performing data flow testing on classes. In *Proc. of the 2nd Symp. on the Foundations of Software Engineering* (1994), pp. 154–163.

[6] KUNG, D. C., GAO, J., HSIA, P., TOYOSHIMA, Y., AND CHEN, C. On regression testing of object-oriented programs. *J. of Systems and Software 32* (1996), 21–40.

[7] MARLOWE, T. J., AND RYDER, B. G. An efficient hybrid algorithm for incremental data flow analysis. In *Proc. of the ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages* (Jan. 1990), pp. 184–196.

[8] PERRY, D. E., AND KAISER, G. E. Adequate testing and OO programming. *J. of Object-Oriented Programming* (1990).

[9] ROTHERMEL, G., AND HARROLD, M. J. Selecting regression tests for object-oriented software. In *Proc. of the Int. Conf. on Software Maintenance* (1994).

[10] ROTHERMEL, G., AND HARROLD, M. J. A safe, efficient regression test selection technique. *ACM Trans. on Software Engineering and Methodology 6*, 2 (April 1997), 173–210.

[11] SOUTER, A., AND POLLOCK, L. Omen: A strategy for testing object-oriented software. In *Proc. of ACM SIGSOFT 2000 Int. Symp. on Software Testing and Analysis (ISSTA)* (August 2000), p. 49'59.

[12] TIP, F. A survey of program slicing techniques. *J. of Programming Languages 3*, 3 (1995), 121–189.

[13] TIP, F., AND PALSBERG, J. Scalable propagation-based call graph construction algorithms. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)* (Minneapolis, MN, 2000), pp. 281–293. *SIGPLAN Notices* 35(10).

[14] WEYUKER, E. Axiomatizing software test data adequacy. *IEEE Trans. on Software Engineering SE12:12* (1986), 668–675.

[15] ZELLER, A. Yesterday my program worked. Today, it does not. Why? In *Proc. of the 7th European Software Engineering Conf./7th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE'99)* (Toulouse, France, 1999), pp. 253–267.