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 [12] 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\footnote{See www.ibm.com/software/ad/vajava}. 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 I(a) will be used as a running example to illustrate our notion of change impact analysis. The example consists of five classes:

\footnote{On sabbatical at IBM T.J. Watson Research Center 2000-2001 from Rutgers University.}

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

\textit{PASTE} '01, June 18-19, 2001, Snowbird, Utah, USA. Copyright 2001 ACM 1-58113-413-4/01/0006 ...$5.00.
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 getTotalCredits() {
        int sum = 0;
        for (Course c : courses) sum += c.getcredits();
        return sum;
    }
    public String toString() { return Student + courses; }
}

class Course {
    private String id;
    private int credits;
    private Student[] students = new Student[10];
    private int size = 0;
    public Course(String id, int credits) {
        this.id = id;
        this.credits = credits;
    }
    public void addStudent(Student s) {
        students[size++] = s;
    }
    public int size() {
        return size;
    }
    public String toString() {
        return id + size; // return information about the course
    }
}

class University {
    private Set professors;
    University() { professors = new HashSet();
    }
    public void addProfessor(Professor p) {
        professors.add(p);
    }
    public Professor findProfessor(String name) {
        for (Professor p : professors)
            if (p.getName().equals(name))
                return p;
        return null;
    }
    public String toString() { return University; }
}

class TestA {
    public static void main(String arg[]) { new University();
        Professor p1 = new Professor("Barbara Ryder", "DSN", "CORE 311");
        addProfessors(p1);
        addProfessor(p1);
        Professor p2 = new Professor("Frank Tipt", "DSN", "CORE 300");
        addProfessors(p2);
        Course c1 = new Course("100", p1);
        Professor p3 = new Professor("Donald Smith");
        if (p == null)
            System.out.println("Professor Donald Smith found");
    }
}

class TestB {
    public static void main(String arg[]) { new University();
        addProfessors(p1);
        addProfessor(p1);
        Professor p = new Professor("Barbara Ryder");
        if (p != null)
            System.out.println("Professor Barbara Ryder found");
    }
}

class TestC {
    public static void main(String arg[]) { new University();
        addProfessors(p1);
        addProfessor(p1);
        Student s1 = new Student("Matt Arnold");
        addStudent(s1);
        Student s2 = new Student("Ann Milanova");
        addStudent(s2);
    }
}

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 forms 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 run 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 class 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

| (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.

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.

### 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) adding any kind of statement-level changes inside a method body. The LC category “abstracts” any kind of source code change that affects dynamic dispatch behavior.

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 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-...
\[ \text{Lookup} = \{ (C, A.m, B.m) \mid \text{class } A \text{ contains virtual method } m, C \leq B \leq A, \]
\[ \text{class } B \text{ contains virtual method } m, \]
\[ \text{there is no class } B' \text{ that contains method } m \text{ such that } C \leq B' \leq A \} \]

\[ \text{LC} = \{ \{a, b\} \mid \{a, b, c\} \in ((\text{Lookup}_{\text{old}} - \text{Lookup}_{\text{new}}) \cup (\text{Lookup}_{\text{new}} - \text{Lookup}_{\text{old}})) \} \]

\[ \text{SameLookup}(B, f()) = \{ \langle C, Y.f(), V.f() \rangle \mid \langle B, Y.f(), V.f() \rangle \in \text{Lookup}_{\text{old}}, \langle C, Y.f(), V.f() \rangle \in \text{Lookup}_{\text{old}}, C \leq B \leq Y \} \]

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

...arguments, 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 \( \langle \text{runtimeReceiverType, staticMethodSignature, actualMethodBound} \rangle \), then some of the values corresponding to our original hierarchy of Figure 1 are:

\[
\begin{align*}
\text{Person, Person.toString()} & \text{, Person.toString()} \\
\text{Student, Person.toString()} & \text{, Person.toString()} \\
\text{Professor, Person.toString()} & \text{, Professor.toString()} \\
\text{Person, Person.getName()} & \text{, Person.getName()} \\
\text{Student, Person.getName()} & \text{, Person.getName()} \\
\end{align*}
\]

Using the inheritance operators of Table 2, the complete Lookup map can be defined as the set of all tuples \( \langle C, A.m, B.m \rangle \) 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 \( \text{Lookup}_{\text{old}} \) and \( \text{Lookup}_{\text{new}} \) to refer to the set of Lookup tuples before and after the edit, respectively. We can now define LC as a set of pairs \( \langle C, A.m() \rangle \), 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 invalid tuples from Lookup, and adding some newly created tuples. For example, when a method \( B.f() \) is added, we may remove some tuples from \( \text{Lookup}_{\text{old}} \) 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 \( B.f() \).

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

\[
\begin{align*}
A < B & \text{: } A \text{ is a direct descendent of } B \\
A \leq B & \text{: } A \text{ is a direct descendent of } B, \text{ or } A = B \\
A \leq \star B & \text{: } B \text{ is an ancestor of } A, \text{ or } A = B \\
A < \star B & \text{: } B \text{ is an ancestor of } A, \text{ but } B \neq A
\end{align*}
\]

**Table 2: Notation for inheritance relations.**

...
<table>
<thead>
<tr>
<th>edit</th>
<th>deleted tuples</th>
<th>added tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>add method $B.f()$</td>
<td>SameLookup($B.f()$)</td>
<td>$(C \cdot X \cdot f(), B \cdot f()) \in \text{SameLookup}(B, f(), C \leq B &lt; ^* Y &lt; ^* X)$</td>
</tr>
<tr>
<td>delete method $B.f()$</td>
<td>$(C \cdot X \cdot f(), B \cdot f()) \mid C \leq B \leq ^* A$, $D \leq ^* C &lt; B$</td>
<td>$(C \cdot X \cdot f(), B \cdot f()) \in \text{Lookup}(B, f(), C \leq B &lt; ^* A)$</td>
</tr>
<tr>
<td>add leaf class $C$</td>
<td>none</td>
<td>$(C \cdot X \cdot f(), V \cdot f()) \in \text{SameLookup}(B, f(), C &lt; B)$</td>
</tr>
<tr>
<td>delete leaf class $C$</td>
<td>none</td>
<td>$(C \cdot X \cdot f(), V \cdot f()) \in \text{SameLookup}(B, f(), C &lt; B)$</td>
</tr>
<tr>
<td>move nonempty leaf class $B$</td>
<td>$(B, X \cdot f(), V \cdot f()) \in \text{Lookup}$</td>
<td>$(B, X \cdot f(), V \cdot f()) \in \text{Lookup}$</td>
</tr>
<tr>
<td>move subtree rooted at $B$</td>
<td>$(C, X \cdot f(), V \cdot f()) \in \text{Lookup}$</td>
<td>$(C, X \cdot f(), V \cdot f()) \in \text{Lookup}$</td>
</tr>
</tbody>
</table>

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:

```java
(Professor, Person.toString(), Professor.toString())
(Professor, Professor.toString(), Professor.toString())
```

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

```java
(Professor, Person.toString(), Person.toString())
```

Hence, we have that:

$$
\text{LC} = \{ (\text{Professor}, \text{Person.toString}(), \text{Professor.toString()}),
(\text{Professor}, \text{Professor.toString}(), \text{Professor.toString}())
\}$$

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 GradStudent and Undergrad to Figure 1, then we will have GradStudent < Student and GradStudent < 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 GradStudent and Undergrad:

```java
(GradStudent, Person.toString(), Person.toString())
(GradStudent, Person.getName(), Person.getName())
(GradStudent, Student.addCourse(), Student.addCourse())
(GradStudent, Student.totalCredits(), Student.totalCredits())
(Undergrad, Person.toString(), Person.toString())
(Undergrad, Person.getName(), Person.getName())
(Undergrad, Student.addCourse(), Student.addCourse())
(Undergrad, Student.totalCredits(), Student.totalCredits())
```

Hence, the impact on LC is:

$$
\text{LC} = \{ (\text{GradStudent}, \text{Person.toString}()),
(\text{GradStudent}, \text{Person.getName}(), \text{Person.getName}())
(\text{GradStudent}, \text{Student.addCourse}(), \text{Student.addCourse}())
(\text{GradStudent}, \text{Student.totalCredits}(), \text{Student.totalCredits}())
\}$$

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 GradStudent and Undergrad of Section 3.2.3 as the basis for the next example. Here, we consider the deletion of class Undergrad. This implies that all tuples whose run-time type component is Undergrad will be removed from Lookup, i.e.:

```java
(Undergrad, Person.toString(), Person.toString())
(Undergrad, Person.getName(), Person.getName())
(Undergrad, Student.addCourse(), Student.addCourse())
(Undergrad, Student.totalCredits(), Student.totalCredits())
```

Hence, the impact on LC is:

$$
\text{LC} = \{ (\text{Undergrad}, \text{Person.toString}()),
(\text{Undergrad}, \text{Person.getName}(), \text{Person.getName}())
(\text{Undergrad}, \text{Student.addCourse}(), \text{Student.addCourse}())
(\text{Undergrad}, \text{Student.totalCredits}(), \text{Student.totalCredits}())
\}$$

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.

```java
(GradStudent, Person.toString(), Person.toString())
(GradStudent, Person.getName(), Person.getName())
(GradStudent, Student.addCourse(), Student.addCourse())
(Undergrad, Person.toString(), Person.toString())
(Undergrad, Person.getName(), Person.getName())
(Undergrad, Student.addCourse(), Student.addCourse())
(Undergrad, Student.totalCredits(), Student.totalCredits())
```
and therefore, the changes to Lookup will consist of adding tuples to represent those functions inherited from Person:

{ Teacher, Person.getName(), Person.getName() }  
{ Teacher, Person.toString(), Person.toName() }, 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, LookupOld contains the following tuple due to inheritance from Person:

{ Professor, Person.getName(), Person.getName() }  

When Professor becomes a child of Teacher, we must remove this tuple. Then, we need to add tuples to LookupNew 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:

{ Professor, Teacher.numCourses(), Teacher.numCourses() }  
{ Professor, Person.getName(), Person.getName() }, etc.

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

{ Professor, Person.getName(), Person.getName() }  

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:

{ Professor, Professor.toString(), Professor.toString() }  

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 ≤ on atomic changes (with transitive closure ≤*). For a given set A of atomic changes that transforms P into P', ≤ 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:

∀a ∈ A such that a ≤ a', a ∈ A' ⇒ a' ∈ 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 upsource 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: c1 ≡ Student.idNum ∈ AF, c2 ≡ Student.StudID ∈ AM, c3 ≡ Student.String() ∈ AC, and c4 ≡ Person.toString() ∈ AC. In this example, we have that c1 ≤ c2, c1 ≤ c3 ≤ c4, and c3 ≤ c4.

The second edit allowed each person to be affiliated with a department. This edit corresponds to the following atomic changes: c5 ≡ Person.department ∈ AF, c6 ≡ Professor.department ∈ DF, c7 ≡ Person.Person(String,String) ∈ AM, and c8 ≡ Person.toString() ∈ AC. In this example, we have that c5 ≤ c6, c6 ≤ c7 ≤ c8, and c7 ≤ c8.

The third edit implements the new rule that caps course enrollment at 50 students. This corresponds to one atomic change, c9 ≡ University.enrollInCourse() ∈ CM.

4. CHANGE IMPACT ANALYSIS

We will assume that associated with program P is a set of test drivers T = t1, · · · , tk. Each test driver ti exercises a subset Nodes(P, ti) of P's methods, and a subset Edges(P, ti) of calling relationships between P's methods. Likewise, NodeS(P, ti) and Edges(P, ti) form the call graph for ti on the edited program P'. Here, a calling relationship between methods is assumed to be of the form A.m → 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.
Figure 3 shows definitions of the two key concepts that form the foundation of our analysis. AffectedTests($T,A$) is a subset of $T$ containing only those test drivers whose behavior may be affected by changes in $A$. 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($t,A$) is a subset of the changes in $A$ 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 $Edges^a$. We plan to experiment with efficient call graph construction algorithms such as RTA [1] and XTA [33], 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($T,A$) is guaranteed to produce the same result after incorporating the changes in $T$. Hence, only test cases in AffectedTests need to be re-executed and have their results examined by the programmer.
- Affecting Changes 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.
- Affecting Changes can provide useful information once a test driver has failed, by allowing the programmer to focus on failure-related changes.

Let $T = \{ TestA, TestB, TestC \}$. 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($T$, $\{ c_1, c_2, c_3, c_4 \}$), because the method changed by $c_2$ (the constructor of class Student) occurs in $Nodes(P, TestB)$ and in $Nodes(P, TestC)$. 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(TestA, $\{ c_1, \ldots, c_{14} \}$) $\leq \{ c_7, c_8, c_{10}, c_{11}, c_{12}, c_{13} \}$. 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

In these formulae, assume that AM, DM, CM are encoded as nodes in the call graph of $P$ or $P'$. 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 interdependencies 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
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 [6] 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