CRISP - A FAULT LOCALIZATION TOOL FOR JAVA PROGRAMS

BY OPHELIA C. CHESLEY

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Barbara G. Ryder

and approved by

New Brunswick, New Jersey

October, 2007

ABSTRACT OF THE THESIS

Crisp - A Fault Localization Tool for Java Programs

by Ophelia C. Chesley

Thesis Director: Barbara G. Ryder

Crisp is a tool (i.e., an Eclipse plug-in) for constructing intermediate versions of a Java program that is being edited in an IDE such as Eclipse. After a long editing session, a programmer usually will run regression tests to make sure she has not invalidated previously checked functionality. If a test fails unexpectedly, *Crisp* uses input from *Chianti*, a tool for semantic change impact analysis [13], to allow the programmer to select parts of the edit that affected the failing test and to add them to the original program, creating an intermediate version guaranteed to compile. Then the programmer can re-execute the test in order to locate the exact reasons for the failure by concentrating on those affecting changes that were applied. Using *Crisp*, a programmer can 1) iteratively select, apply, and *undo* individual (or sets of) affecting changes and, thus effectively find a small set of failure-inducing changes, or 2) request *Crisp* to automatically suggest a set of failure-inducing changes.

Acknowledgements

Special thanks to my advisor, Dr. Barbara G. Ryder, whose devotion to education and research prompted my initial interest in programming languages, program analysis and software engineering. Also, I would like to thank my research colleagues Xiaoxia Ren and Maximilian Stoerzer for their hard work, collaboration and valuable suggestions. Last but not least, I would like to thank my husband, Dennis, for his support and patience.

Table of Contents

Chapter 1

Introduction

Software evolves over time due to user requirements changes, feature enhancements, and bugs fixes. In a development environment, it is common for programmers to test their edited version of the software against a set of regression tests to ensure that the changes they made do not introduce unexpected behavior in the software. When a test fails, it is the programmer's responsibility to locate the failure as well as the changes that may have caused the failure. For small sized software, an interactive debugger is usually used to trace the execution path of the failed test. This process requires programmer's knowledge of both the edit and the source code of the software. It does not scale to medium or large software for the following reasons:

1. The combined size of the edit could be large with different groups of developers working on different components of the software.

2. Components within a piece of software are usually connected by programming interfaces. These interfaces could be misused due to the lack of, or poor documentation. Very often, each component is tested separately with its own unit testing, while regression testing is performed after integrating all the changed components into the previously tested version of the software. It is only through vigorous regression testing that the proper use of these interfaces can be confirmed. When a test fails during this phase, it is difficult to identify all the changed components that have been touched by the test and to sort out which components require further analysis.

3. For large software development, regression testing could be performed by a group of testers that are not familiar with the source code. When a test that exercises

different components of the software fails, it would be beneficial for the testers to identify the problematic portion of the edit and the corresponding components. This could facilitate the distribution and communication of the results to the responsible developers.

4. When using current object-oriented programming languages, source code edits could have non-local effects during execution due to dynamic dispatch of virtual methods based on the run-time type of the receiver object. Any addition or deletion of virtual methods as well as changes in the type hierarchy could have rippling effects beyond the original intent of the changes.

5. Statement-level tracing and analysis commonly used in debuggers and other fault finding techniques are not scalable once the software grows beyond a few thousand lines of code. An edit needs to be modeled at a granularity that is effective for analysis, yet safe and efficient to provide interactive feedbacks to the users.

Crisp is an interactive tool developed as an Eclipse plugin to identify failure-inducing changes for Java programs. *Crisp* is an extension to *Chianti* [13], a change impact analysis prototype that provides safe and summarized information pertaining to a set of tests. *Chianti* decomposes an edit into a set of *atomic changes* and calculates the subset of these atomic changes, namely *affecting changes*, that are exercised by each test. These atomic changes, or affecting changes, model changes at the class, field, and method levels of the Java constructs. This allows the analysis to provide timely interactive feedback to users. Initial experimental results on *Chianti* using several medium to large scale software projects are promising. On average, changes that are related to a test constitute only 3.95% of the size of the edits.

When a test fails in the edited version of the software, the set of affecting changes provided by *Chianti* can be reviewed by the programmer to identify the suspicious ones that might have caused the failure. However, the number of affecting changes for the failing test can be large. Without *a priori* knowledge of the total number of failure-inducing changes, following a systematic approach to identify all of them can be time consuming. Furthermore, once the suspicious changes have been identified, there is no efficient approach of confirming that they are indeed failure-inducing without falling back to the traditional debugging process of statement-level analysis.

Crisp is developed to provide an alternative choice to the situation above. Once the suspicious affecting changes are identified, *Crisp* can extract these changes from the edited version and apply them to the original version of the software, simply by adding the changes or by replacing the original constructs with the edited ones. The resulting version, namely the *intermediate* version, is neither original nor edited. When a failing test is executed against this intermediate version and continues to fail, the failure-inducing changes are among those that have been applied. Otherwise, there are failure-inducing changes among those that have not been applied. By iteratively exploring and observing the test outcomes on intermediate versions created by applying subsets of affecting changes, programmers are able to focus on the few that cause the test to fail. Furthermore, in the event that knowledgeable programmers are not available or there are too many affecting changes, *Crisp* can automatically suggest a set of failure-inducing changes with minimal user input.

The key to the above iterative approach is the creation of valid intermediate versions. These intermediate versions need to be free of compilation errors, or else the users would have to edit them before continuing their exploration. To ensure compilability, *Crisp* requires that *Chianti* annotates each atomic change with its relationships to, or *dependences* on other atomic changes during the decomposition of the edit. Using these dependences, *Crisp* calculates sets of changes that can be applied to the original program to create valid intermediate versions of the software program.

Overall, the contribution of *Crisp* encompasses the provision of an interactive environment that:

1. Organizes affecting changes and their dependences to facilitate the creation of compilable intermediate versions;

2. Provides the programmers with a graphical user interface where they can select affecting changes of interest to explore various intermediate versions semi-automatically;

3. Includes programmable interfaces where affecting changes can be ranked, based on different heuristics, in terms of their likelihood of being the failure-inducing changes. This provides additional guidance to programmers who may not be familiar with the development of all the components within the software;

4. Automatically generates for each failure-inducing changes set, a complementary set. This feature is necessary to confirm the completeness of a failure-inducing changes set;

5. Upon requests from users, often can locate failure-inducing changes automatically using an exploration algorithm.

Chapter 2

Background

2.1 Change Impact Analysis

In order to locate failure-inducing changes for a failing test, *Crisp* relies on *Chianti* to provide source code change information that is relevant to a test. This chapter is an overview of the aspects of *Chianti* that are relevant to *Crisp*. Detailed explanation of the implementation and experimental results of *Chianti* can be found in [13]. The conceptual design of the analyses in *Chianti* can be found in [15].

In summary, *Chianti*:

1. Takes as input an original version and an edited version of a program and a test suite that exercises the program code. It is assumed that the test suite can run on both versions of the program.

2. Decomposes the edit between the original and the edited version into atomic changes of types shown in Figure 2.1 by comparing their *abstract syntax trees*.

3. For every test in the test suite, uses its original call graph (which contains all the calls when the test is executed against the original version) to determine whether the test is *affected* by the edit or not.

4. For each *affected test*, uses its edited call graph (which contains all the calls when this test is executed against the edited version) to calculate a set of *affecting changes*. All the changed methods **CM** that correspond to the nodes in this call graph and lookup changes **LC** that correspond to the edges in the graph are considered affecting changes to the test.

Based on their definitions, affecting changes are a subset of atomic changes that are relevant to a test. Specifically, affecting changes that are generated by using a dynamic call graph, constructed by tracing the execution of a test, are in fact all the changes that are exercised by the test. If this test fails, examining its affecting changes could be the first step in the debugging process.

Category	Notation	Explanation
AC	**AC**(A)	Add an empty class A
DC	**DC**(A)	Delete an empty class A
CTD	**CTD**(A)	Change type declaration/hierarchy of class A
AM	**AM**(A.foo())	Add an empty method foo() in class A
DM	**DM**(A.foo())	Delete an empty method foo() in class A
CM	**CM**(A.foo())	Change the body of a method foo() in class A
LC	**LC**<B, A.foo()>	Change the virtual method lookup of a run-time receiver of type B that calls A.foo()
AF	**AF**(A.x)	Add a field x in class A
DF	**DF**(A.x)	Delete a field x in class A
CFI	**CFI**(A.x)	Change the definition of an instance field x's initializer in class A
CSFI	**CSFI**(A.x)	Change the definition of a static field x's initializer in class A
AI		Add an empty instance initializer
DI		Delete an empty instance initializer
CI		Change the definition of an instance initializer
ASI		Add an empty static initializer
DSI		Delete an empty static initializer
CSI		Change the definition of an static initializer

Table 2.1: Categories of atomic changes.

Each of the 17 types of atomic changes, except **LC**, has a direct connection to a source code change between the original and the edited versions of the program. **LC** models a change in the dynamic dispatch of a virtual method during runtime due to an addition or a deletion of a method within a type hierarchy as well as a change of the type hierarchy. Assume a method is added to an original program consisting of a single public class A with a non-private method foo(). In the edited version, another non-private method bar() is added in A. *Chianti* reports two atomic changes based on the source code change: **AM**(A.bar()) for declaring a new method A.bar() and **CM**(A.bar()) for implementing the method body. In addition, at runtime, the edited version could have a receiver object of type A that calls the method bar(). In this case, A.bar() is executed. This additional possible dispatch of A.bar() for a run-time receiver of type A is modeled by an atomic change **LC**<A,A.bar()>.

Here is another example to illustrate a change of the type hierarchy. Again, the original program consists of a single public class A with a non-private method foo(). In the edited version, a class B is added as a subclass class of A. *Chianti* reports an atomic change **AC**(B). In addition, at runtime, the edited version could have a receiver object of type B that calls the method foo(). In this case, A.foo() is executed. This additional possible dispatch of A.foo() for a run-time receiver of type B is modeled by an atomic change **LC**<B,A.foo()>. Other **LC** changes include changing a *private* method that is originally not dispatched dynamically to *public*; making an *abstract* class C *concrete* such that a run-time receiver object could be of type C. In all of these examples, the **LC** change is a result of a source code change that can have a run-time impact.

Other than modeling a change in run-time dispatch behavior, *Chianti* also inserts special **AM** and **CM** changes to model two initializers that do not correspond to any source code edit. The Java compiler creates a no-argument implicit constructor in the absence of an explicit constructor in a class. This no-argument implicit constructor is called, at runtime, when the object is instantiated. Since this implicit constructor, named <init>, may be present in the call graphs of a test, it is essential for *Chianti* to capture it as an atomic change. Therefore, if class A is added in the edit without an explicit constructor, *Chianti* generates two changes: **AC**(A) and **AM**(A.<init>()). Similarly, if class A contains an instance variable x of type integer with an initial value 10, *Chianti* generates three more changes: **AF**(A.x), **CFI**(A.x), and **CM**(A.<init>()). Besides the implicit constructor, the Java compiler also inserts a method named <clinit> for class variables initialization. If class A also contains a class variable c of type String with an initial value of “How are you?”, *Chianti* generates four changes: **AF**(A.c), **CSFI**(A.c), **AM**(A.<clinit>()), and **CM**(A.<clinit>()). In summary, the insertion of atomic changes for <init>() and <clinit>() in *Chianti* guarantees that any source code edit pertaining to initialization can be mapped to the call graphs generated during the execution of a test. This is critical to the calculation of affected tests and their resulting affecting changes.

2.2 Atomic Changes and Their Dependences

In order to allow programmers to examine the effects of certain affecting changes on a failing test, *Crisp* applies selected affecting changes to the original program. The resulting intermediate version, however, may not compile. In order to ensure compilability, *Crisp* uses the dependences among the atomic changes that are captured by *Chianti*. An atomic change A_1 is a *prerequisite* of A_2 (denoted by $A_1 \prec A_2$) if when a programmer selects to apply A_2 to the original program, A_1 needs to be applied also in order for the intermediate version to be compilable. In other words, A_2 is dependent on A_1. For a complete discussion on dependences, please refer to [5] and [12]. The four kinds of dependences used by *Crisp*: structural, buddy, declaration, and mapping, are explained below. This is a summary of the discussion on dependences found in [5] and [12].

2.2.1 Structural Dependences

Intuitively, structural dependences capture the necessary sequences that occur when Java elements are added or deleted in a program. When A_2 has a structural prerequisite A_1 ($A_1 \prec_{Structural} A_2$), *Crisp* has to apply A_1 to the original program first, or else it will be impossible to add A_2 later. A simple example of this relationship is that adding a new class **AC**(A) has to occur before adding a new member field **AF**(A.x) or a new member method **AM**(A.foo()). Hence **AC**(A) $\prec_{Structural}$ **AF**(A.x) and **AC**(A) $\prec_{Structural}$ **AM**(A.foo()). Since Java allows anonymous classes and local classes be added or deleted within a method, the method needs to exist in the program before adding the classes it encloses. For example, if a new method A.foo() is added in which a local class C is defined in the method body, then **AM**(A.foo()) $\prec_{Structural}$ **AC**(A\$1\$C). Finally, structural dependences are created when there is a change in the definition of a field or method. A definition change is handled in *Chianti* by first deleting the old definition and then adding the new definition. Hence, *Chianti* decomposes a type change of a field into two atomic changes: **DF** and **AF**. Similarly, *Chianti* decomposes a return type change of a method into two atomic changes: **DM** and **AM**. In order to ensure that the intermediate version never contains two fields or two methods with the same name but different definitions, structural dependences are used, namely **DF** $\prec_{Structural}$ **AF** and **DM** $\prec_{Structural}$ **AM**.

2.2.2 Buddy Dependences

A buddy dependence is essentially a form of structural dependence. However, *Crisp* handles buddy dependences in a different way than other structural dependences. A full discussion on how *Crisp* uses them to construct intermediate versions can be found in Chapter 3. When an atomic change A_1 is a buddy prerequisite to A_2 ($A_1 \prec_{Buddy} A_2$), they are *inseparable.* Applying A_2 requires A_1 because of a structural reason; in addition, applying A_1 requires A_2 because of a compilation reason. For example, an added method A.foo() generates two buddies: **AM**(A.foo()) and **CM**(A.foo()). The body of A.foo() cannot be applied until after the declaration of A.foo(). However, the declaration of A.foo() may not compile without the presence of the body due to a non void return type of A.foo(). As a result, these two atomic changes have to be applied together. There are many kinds of buddy dependences used in *Crisp*; a complete list can be found in Table 3.1 of Chapter 3.

2.2.3 Declaration Dependences

In general, any Java element declarations that are required to create valid intermediate versions are captured in this category. A_1 is a declaration prerequisite of A_2 ($A_1 \prec_{Declaration} A_2$) generally means that A_2 uses the Java element declared in A_1 in its source code. An example is that a newly added method A.foo()'s body uses a newly added field A.x. This field must be declared in the intermediate version for it to compile. Hence, **AF**(A.x) $\prec_{Declaration}$ **CM**(A.foo()). Other examples of declaration dependences include adding a new class B that is a subclass of another new class A requires the declaration of A (**AC**(A) $\prec_{Declaration}$ **AC**(B)); using a new type A as a return type or a parameter in a new method X.foo() (**AC**(A) $\prec_{Declaration}$ **AM**(X.foo()).

Declaration dependences are also necessary to ensure the presence of concrete method implementations within a type hierarchy that contains abstract methods. An abstract method foo() cannot be added to an abstract class A (or interface A) unless all of A's subclasses add a concrete method foo(). Assume A has only one concrete subclass B to begin with. Then, **AM**(B.foo()) $\prec_{Declaration}$ **AM**(A.foo()). If a new class C is also added which is a subclass of A, it needs to implement C.foo() in order for the edit to compile. This results in **AM**(C.foo()) $\prec_{Declaration}$ **AC**(C). [1] Remember that there is also a structural dependence between C and C.foo() (**AC**(C) $\prec_{Structural}$ **AM**(C.foo())). These two dependences combined guarantee there will never be an intermediate version that does not contain both class C and method C.foo() together. Similarly, B.foo() cannot be deleted without A.foo() being changed to a concrete method. In this case, **CM**(A.foo()) $\prec_{Declaration}$ **DM**(B.foo()).

2.2.4 Mapping Dependences

The above 3 types of dependences are generated by syntactic relationships among parts of an edit. Since *Chianti* also captures dynamic dispatch and initializer methods that are in addition to explicit source changes, it needs a mechanism to *map* them to the corresponding source code changes. This gives rise to mapping dependences. Informally, an atomic change A_1 is a mapping prerequisite to A_2 ($A_1 \prec_{Mapping} A_2$) means that the source code change generated by A_1 results in A_2. In Section 2.1, an **LC** change can be generated by an added method or a change in the type hierarchy. Hence, **AM**(A.bar()) $\prec_{Mapping}$ **LC**<A,A.bar()> when a new non-private method A.bar() is added. Similarly, **AC**(B) $\prec_{Mapping}$ **LC**<B,A.foo()> when a new subclass B of A is added. For initializer methods, **AC**(A) $\prec_{Mapping}$ **AM**(A.<init>()) is generated when a new class A is added without an explicit constructor; **CFI**(A.x) $\prec_{Mapping}$ **CM**(A.<init>()) when there is a changed in the initialized value of an instance variable A.x. When the first class variable A.s with an initialized value of "How are you?" is added, the <clinit>() initializer method is created. This results in **CSFI**(A.s) $\prec_{Mapping}$ **AM**(A.<clinit>())

[1] Note that A.foo() does not depend on C.foo()

as well as **CSFI**(A.s) $\prec_{Mapping}$ **CM**(A.<clinit>()) begin generated.

When *Chianti* reports a set of affecting changes of an affected test, it includes the affecting changes exercised by the affected test as well as the transitive closure of these changes' prerequisites based on the dependences [13]. This initial set is used by *Crisp* to create intermediate version. In addition, the design of *Crisp* also has to take into consideration the granularities of the changes that are applied, the positions of source code additions, changes to the `import` statements in a Java class, and the usability of the tool itself in an interactive environment.

Chapter 3

Design of *Crisp*

3.1 Limitations of Affecting Changes

Crisp is written with three major application modes in mind: (i) an automatic mode where failure-inducing changes are calculated and presented to programmers for inspection (see Chapter 5 for details), (ii) an interactive session where programmers are free to explore by selecting one or more affecting changes to be applied to the original program and re-execute the failing tests in many different intermediate versions, and finally, (iii) a batch mode where a set of changes can be applied to the original program to create a particular intermediate version. Since *Chianti* already provides all the dependences among the atomic changes, it is relatively direct to obtain a prerequisites set for one or more affecting changes by traversing the dependence graph. These prerequisites sets in theory contain only affecting changes, since they are the changes that are executed by an affected test, as well as the transitive closure of their prerequisites, as explained in Section 2.2.

Chianti's definitions for affecting changes sets, although comprehensive, are insufficient when it comes to generating compilable intermediate versions of Java programs. First, *Chianti* very often generates more than one atomic change for each unit of source code change. This potentially introduces unexpected compilation errors in the intermediate versions. Second, *Chianti* reveals redundant affecting changes information to programmers in the interactive mode which can affect the usability of *Crisp*. Third, applying each affecting change means updating a source file or compilation unit in Java. Such a file has to exist in the program before it can be updated. Therefore, a specific ordering is needed when applying an affecting change and its prerequisites.

To guarantee the creation of compilable intermediate versions, *Crisp* takes the affecting changes set generated by *Chianti* and creates a corresponding *to-be-applied* list for each affecting change. The elements within these lists are still referred to as prerequisites, but they can be either affecting or non-affecting changes. The prerequisites in each list have a pre-determined order and are therefore applied to the original program according to this order. Before presenting them to the user, *Crisp* sweeps through the affecting changes and their to-be-applied lists and eliminates all the redundant affecting changes and prerequisites. The next few sections of this chapter provide the details of the processes in which *Crisp* creates the to-be-applied lists for the affecting changes. Chapter 4 describes the presentation of changes in *Crisp* and the construction of valid intermediate versions.

3.2 Prerequisites from *Chianti* to *Crisp*

The shortcomings of using affecting changes and their prerequisites for creating intermediate versions can be illustrated using an example. Figure 3.1(a) shows a changed program. The boxed text indicates program codes that have been added in the edited version. The struck out program codes are those that have been deleted. Each piece of code change results in one or more affecting changes, indicated by the numbers associated with each piece of boxed text. The test case TestSimple.test() passes in the original version and fails in the edited version. *Chianti* generates the dependence graph shown in Figure 3.1(b). There are eight affecting changes for this failing test, and they are connected by four types of arrows representing the categories of dependences described in Section 2.2. [1] Essentially, Figure 3.1 illustrates a simple example of *Chianti*'s input and output. Using the affecting changes and dependences in Figure 3.1(b), *Crisp* generates the prerequisites sets and eventually the to-be-applied lists in Figure 3.2. In Figure 3.2(a), each affecting change and its corresponding prerequisites set are calculated by simply traversing the dependence graph. Note that the affecting change itself is included in the prerequisites set. This table mimics the lookup table maintained in

[1]Other non-affecting atomic changes are not shown in Figure 3.1

Crisp. When a user selects an affecting change from the left hand column, the prerequisites in the corresponding row in the right hand column would be applied resulting in an intermediate version.

There are several problems if Figure 3.2(a) is presented in the interactive mode and the prerequisites sets are used to apply changes to the original program. As defined by the original atomic changes, a single change does not necessarily represent an entire unit of source code change in the program. For example, **AM**(B.getX()), affecting change 1, has a prerequisites set that contains **AC**(B) and **AM**(B.getX()). Adding an empty method B.getX() results in syntax errors since the method requires the return of a 'String' object. In order to eliminate syntax errors, an arbitrary method body that contains nothing but a 'return null;' statement could be added; however, executing this intermediate version would result in different semantics than the edited version of B.getX().

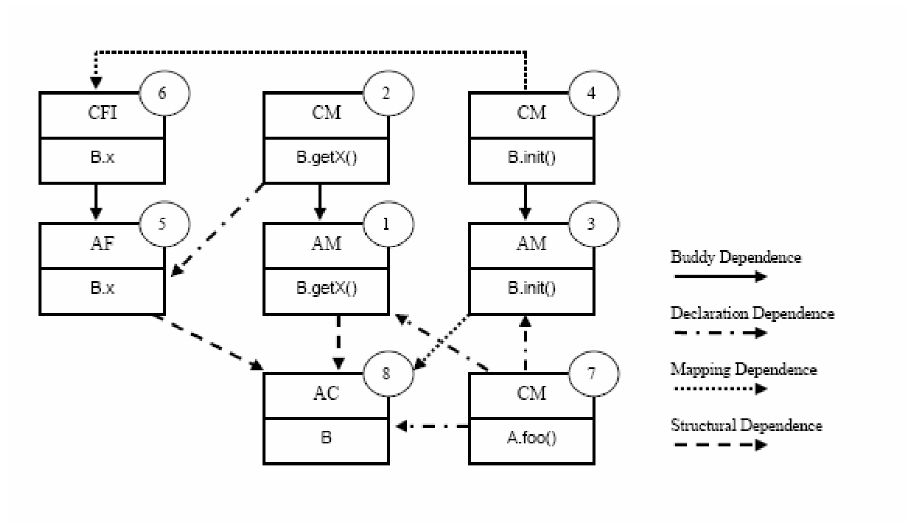
Similarly, B.init() in affecting changes 3 and 4 are used by *Chianti* to represent the implicit constructor that has been added or changed due to the absence of an explicit constructor. As discussed in Section 2.1, these init() (as well as the static implicit initializer, clinit()) methods are not directly related to any visible edits in the source code. Applying **AM**(B.init()), affecting change 3, results in the same intermediate version as if applying **AC**(B), affecting change 1. Similarly, applying **CM**(B.init()), affecting change 4, results in the same intermediate version as if applying **CFI**(B.x), affecting change 6. Hence, their presence in *Crisp* for intermediate version exploration is redundant.

Without ordering, the prerequisites set of **CM**(A.foo()), affecting change 7, could be applied in the order shown in Figure 3.2(a) where **AM**(B.getX()) is added before **AC**(B). Since adding a class usually results in adding a Java file or compilation unit to the program, it is impossible to create the method B.getX() without first creating class B.

Due to the above reasons and more that are explained later, *Crisp* takes the original lookup table of prerequisites sets created by walking the dependence graph, expands it by adding more prerequisites, and then reduces it by deleting some prerequisites as well

```
public class A {
  public String foo() {
    ~~return ‘‘abc’’; }~~
    B b = new B(); return b.getX(); 7
  }
}
public class B { 3,8
  private String x=’’def’’; 4,5,6
  public String getX() { 1
    return x; 2
  }
}
public class TestSimple extends TestCase {
  public void test() {
    A a = new A();
    Assert.assertEquals(a.foo(),’’abc’’); }
}
```

(a) Original and Edited Program:
added code in the edited version appears in boxes;
deleted code is crossed out



(b) Dependence Graph Provided by *Chianti*

Figure 3.1: Affecting Changes and Dependences Generated by *Chianti*

ID	Affecting Changes	Prerequisites sets
1	**AM**(B.getX())	{**AC**(B),**AM**(B.getX())}
2	**CM**(B.getX())	{**AC**(B),**AM**(B.getX()),**AF**(B.x),**CM**(B.getX())}
3	**AM**(B.init())	{**AC**(B),**AM**(B.init())}
4	**CM**(B.init())	{**AF**(B.x),**CFI**(B.x),**AC**(B),**AM**(B.init()),**CM**(B.init())}
5	**AF**(B.x)	{**AC**(B),**AF**(B.x)}
6	**CFI**(B.x)	{**AC**(B),**AF**(B.x),**CFI**(B.x)}
7	**CM**(A.foo())	{**AM**(B.getX()),**AM**(B.init()),**AC**(B),**CM**(A.foo())}
8	**AC**(B)	{**AC**(B)}

(a) Prerequisites Sets Generated by the Dependence Graph

ID	Affecting Changes	To-be-applied lists
1	**AM**(B.getX())	{**AC**(B),**AF**(B.x),**AM**(B.getX())}
5	**AF**(B.x)	{**AC**(B),**AF**(B.x)}
7	**CM**(A.foo())	{**AC**(B),**AM**(B.getX()),**CM**(A.foo())}
8	**AC**(B)	{**AC**(B)}

(b) To-be-applied Lists Generated by *Crisp*

Figure 3.2: Prerequisites Sets and To-Be-Applied Lists Generated by Figure 3.1

as affecting changes, to create a final lookup table with meaningful affecting changes and their corresponding to-be-applied lists. During this process, *Crisp* maintains any necessary ordering among the prerequisites. Figure 3.2(b) shows the resulting lookup table presented by *Crisp*. Notice that all of the to-be-applied lists require the application of **AC**(B) first. Also, the number of prerequisites in the to-be-applied lists may be smaller or larger than their initial prerequisites sets. Finally, only 4 affecting changes are presented to the users, rather than the original 8 of them. The prerequisites in these to-be-applied lists are the ones that are applied to create intermediate versions.

3.3 Expansion of Prerequisites

The very first step when *Crisp* processes the original prerequisites sets from *Chianti* is to order the prerequisites and initialize the to-be-applied lists based on the dependence graphs. Ordering of prerequisites is discussed in Section 3.4. Then *Crisp* checks each prerequisites and decides whether more prerequisites need to be added to the lists. This expansion is necessary to generate semantically accurate intermediate versions and to keep the granularity of changes within the program code at the method level.

3.3.1 Expansion Due to Buddy Changes

For an intermediate version to be semantically meaningful at the method level, an edit of a method has to be applied in its entirety. By contrast, *Chianti* splits the addition of a method into two atomic changes: **AM** denotes adding an empty method; **CM** denotes adding/changing a method body. Similarly, a newly added field with an initialized value should be considered a single edit - yet *Chianti* calculates two separate changes: **AF** denotes declaring a field; **CFI** denotes adding or changing the initial value of the field. To ensure compilability, *Crisp* considers each pair of **AM** and **CM** of an added method inseparable, or *buddy*. *Chianti*, therefore, lists the **CM** as **AM**'s buddy dependent and **AM** as **CM**'s buddy prerequisite. Similarly, for an added field, **AF** is **CFI**'s buddy prerequisite and **CFI** is **AF**'s buddy dependent.

There is also another case of buddy dependences which involves more than a pair of atomic changes. The Java language includes initializer and static initializer blocks within a class in which source code changes could occur. Unlike most of the other source code elements, these blocks do not have names associated with them. In the Eclipse environment, they are numerically represented within the abstract syntax tree according to their relative positions among themselves. This gives rise to ambiguity when there are more than one of these blocks within a class and their relative positions are not consistent between the original and the edited versions. To ensure compilability and the accurate representation of changed source code, *Crisp* considers all of the changes to these blocks within the same class as one single inseparable change. In total, *Crisp* handles 7 buddy change types as shown in Table 3.1.

In most cases, an affecting change's buddy is also an affecting change. Figure 3.2(a) shows the buddy pairs of affecting changes for B.getX(), B.init(), and B.x. Note that the prerequisites set for a buddy dependent contains the buddy prerequisite as well as other changes. To make the buddies inseparable, each prerequisite that has a buddy needs to have this buddy in the to-be-applied lists also. After buddy expansion, these to-be-applied lists include the buddies as well as the transitive closure of the buddy's

```
public class A {
  public String foo() {
    ~~return ‘‘abc’’; }~~
    if (true) return ‘‘def’’; else return getX();} 1
  public String getX() { 2
    return getY();
  }
  public String getY() {
    return ‘‘def’’;
  }
}
public class TestSimple extends TestCase {
  public void test() {
    A a = new A();
    Assert.assertEquals(a.foo(),’’abc’’); }
}
```

(a) Original and Edited Program

ID	Affecting Changes	Prerequisites sets
1	**CM**(A.foo())	{**AM**(A.getX()),**CM**(A.foo())}
2	**AM**(A.getX())	{**AM**(A.getX())}

(b) Prerequisites Sets Provided by Dependence Graph

ID	Affecting Changes	To-be-applied Lists
1	**CM**(A.foo())	{**CM**(A.foo()), **AM**(A.getX()), **CM**(A.getX()), **AM**(A.getY()), **CM**(A.getY())}
2	**AM**(A.getX())	{**AM**(A.getX()),**CM**(A.getX()), **AM**(A.getY()), **CM**(A.getY())}

(c) To-be-applied Lists Generated by *Crisp*

Figure 3.3: An Example of Non-Affecting Changes Included in To-be-applied Lists

Prerequisites	*Dependents*
addMethod **AM**	changeMethod **CM**
changeMethod **CM**	deleteMethod **DM**
addField **AF**	changeFieldInitializer **CFI**
addField **AF**	changeStaticFieldInitializer **CSFI**
changeFieldIntializer **CFI**	deleteField **DF**
changeStaticFieldInitializer **CSFI**	deleteField **DF**
Any one of the initializer block changes **AI**, **CI**, **DI**, **ASI**, **CSI**, **DSI**	All the other initializer block changes in the same class

Table 3.1: Buddy Changes.

prerequisites and their buddies. For example, **AM**(B.getX())'s to-be-applied list should include both **AM**(B.getX()) and **CM**(B.getX()). However, adding **CM**(B.getX()) will not compile unless **AF**(B.x) and its buddy, **CFI**(B.x), are also applied. Hence, the to-be-applied list for **AM**(B.getX()) now contains 5 prerequisites: **AC**(B), **AF**(B.x), **CFI**(B.x), **AM**(B.getX()), and **CM**(B.getX()).

Even though the buddy dependents and their transitive closure prerequisites are now correctly included in the to-be-applied list for each of the affecting changes, these additional changes are not necessarily affecting changes themselves. Consider the program illustrated in Figure 3.3(a). The only affecting changes are those that are numbered, namely 1 and 2. **CM**(A.foo()) is an affecting change because A.foo() has been edited and is exercised when TestSimple.test() is executed. **AM**(A.getX()) is an affecting change because A.getX() is a declaration prerequisite of **CM**(A.foo()) and is within an edit. A.getX() is never executed; the code in its method body cannot affect the outcome of the test. Therefore, the method body of A.get() and the entire method A.getY() are atomic changes of the edit, but are not affecting changes of this test. The original prerequisites sets of the two affecting changes are shown in Figure 3.3(b). Since the affecting change **AM**(A.getX()) is buddy with **CM**(A.getX()), **CM**(A.getX()) needs to be in **CM**(A.getX())'s to-be-applied list. However, **CM**(A.getX()) has a declaration prerequisite, **AM**(A.getY()) which also has a buddy **CM**(A.getY()). After buddy expansion, not only **CM**(A.getX()), a non-affecting change, is added to the to-be-applied lists (Figure 3.3(c)), but also the changes related to A.getY(). It is important to note that these additional atomic changes that are *dragged into* the to-be-applied lists of affecting changes are there only to ensure compilation of the intermediate versions. Programmers who use the interactive mode of *Crisp* are not allowed to select any of these dragged in non-affecting changes as part of their exploration.

3.3.2 Expansion Due to Anonymous Changes

Any sub-method-level changes to anonymous classes of a Java program are captured not only by the changes themselves, but very often by a **CM** change of the enclosing method in *Chianti*. However, the dependence graph generated by *Chianti* is insufficient for creating valid intermediate versions. Figure 3.4(a) shows an example of anonymous changes (changes that are within an anonymous class, including the anonymous class itself) of a program. The original version of class A is shown in Figure 3.4(a)(i) and the edited version is in Figure 3.4(a)(ii) to highlight the fact that the positions of A.bar() and A.foo() have been reversed by the edit. There are two anonymous classes of type C used in A.bar() and A.foo() in Figure 3.4(a)(i). In the abstract syntax tree provided by Eclipse, A\$1 denotes the class C in A.bar() whereas A\$2 denotes the one in A.foo(). Since the anonymous classes are named after their relative positions within a class, A\$1 denotes the class C in A.foo() and A\$2, A.bar() in the edited version, Figure 3.4(a)(ii). The original and edited versions of the remainder of the program can be found in Figure 3.4(a)(iii). Since interface C declares a new method C.get() in the edited version, both the type C anonymous classes used in class A need to implement this get() method. Figure 3.4(b) shows the dependence graph after *Chianti* calculates the atomic changes and affecting changes. Note that the addition of the method C.get() (affecting change 5) in the edited version generated two declaration dependences: 1) the implementation of anonymous C class' get() method in A.foo() and 2) the implementation of anonymous C class' get() method in A.bar(). These dependences ensure that C.get() be only added to the intermediate versions after the concrete types implementing the method, and thus prevent any compilation errors. Out of the eleven atomic changes, only six of them are considered affecting changes(1,5,6,7,8,10). After buddy expansion, the lookup table of the affecting changes is the one shown in Figure 3.5(a). If users are allowed to

```
public class A {
  public void bar() {
    B b = new B();
    b.add(new C() {});}
  public String foo() {
    B b = new B();
    b.add(new C() {});
    return ‘‘abc’’; }
}
```

```
(i) Original Version of class A
```

```
public class A {
  public String foo()[6] {
    B b = new B();
    b.add(new C() {
      public String get() {[1,4,7]
        return ‘‘bcd’’;}[8]});
  return b.get().get();[6] }
  public void bar()[9] {
    B b = new B();
    b.add(new C() {
      public String get() {[2,3,10]
        return ‘‘abc’’;}[11]});}
}
```

```
(ii) Edited Version of class A
```

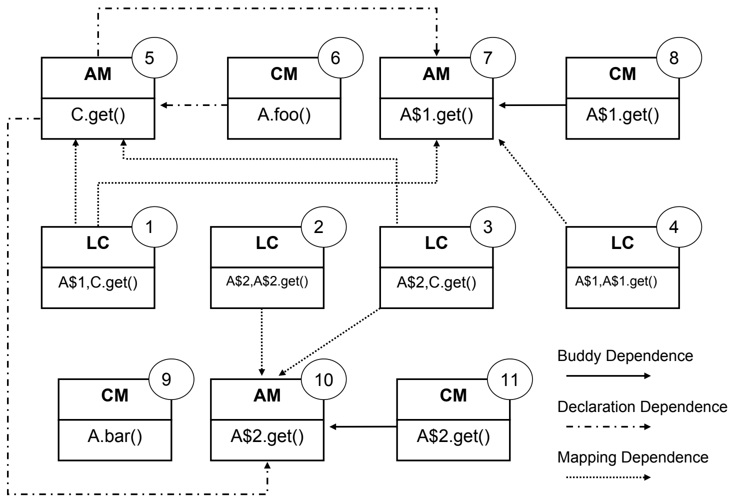
```
public class B {
  private C c;
  public void add(C c) {
    this.c = c;
  }
  public C get() {
    return c;
  }
}

public interface C {
  public String get();[1,3,5]
}

public class TestA extends TestCase {
  public void testA() {
    A a = new A();
    Assert.assertEquals(a.foo(), ‘‘abc’’);
  }
}
```

```
(iii) Original and Edited Versions of Class B
Interface C, and test TestA
```

(a) Original and Edited Versions of Program



(b) Dependence Graph from *Chianti*

Figure 3.4: Anonymous Changes Example

ID	Affecting Changes	To-Be-Applied Lists
5	**AM**(C.get())	{**AM**(A$1.get()), **CM**(A$1.get()),**AM**(A$2.get()), **CM**(A$2.get()),**AM**(C.get())}
6	**CM**(A.foo())	{**AM**(C.get(), **AM**(A$1.get()),**CM**(A$1.get()), **AM**(A$2.get()),**CM**(A$2.get()), **CM**(A.foo())}
7	**AM**(A$1.get())	{**AM**(A$1.get()), **CM**(A$1.get())}
8	**CM**(A$1.get())	{**AM**(A$1.get()),**CM**(A$1.get())}
1	**LC**<A$1,C.get()>	{**AM**(C.get), **AM**(A$1.get()),**CM**(A$1.get()), **AM**(A$2.get()), **CM**(A$2.get()), **LC**<A$1,C.get()>}
10	**AM**(A$2.get())	{**AM**(A$2.get()), **CM**(A$2.get())}

(a) Affecting Changes and To-Be-Applied Lists After Buddy Expansion

```
public String bar() {
  B b = new B();
  b.add(new C() {
    public String get() {
      return ‘‘bcd’’;}
  });
}
```

(b) A.bar() after A$1.get() is applied

```
public String foo() {
  B b = new B();
  b.add(new C() {
    public String get() {
      return ‘‘abc’’;}
  });
}
```

(c) A.foo() after A$2.get() is applied

ID	Affecting Changes	To-Be-Applied Lists
5	**AM**(C.get())	{**AM**(A$1.get()), **CM**(A$1.get()),**CM**(A.foo()), **AM**(A$2.get()), **AM**(A$2.get()),**CM**(A.bar()), **AM**(C.get())}
6	**CM**(A.foo())	{**AM**(C.get(), **AM**(A$1.get()),**CM**(A$1.get()), **AM**(A$2.get()),**CM**(A$2.get()), **CM**(A.bar()), **CM**(A.foo())}
7	**AM**(A$1.get())	{**AM**(A$1.get()), **CM**(A$1.get()),**AM**(A$2.get()), **CM**(A$2.get()), **CM**(A.bar()), **AM**(C.get()), **CM**(A.foo())}
8	**CM**(A$1.get())	{**AM**(A$1.get()),**CM**(A$1.get()), **CM**(A.foo()),**AM**(A$2.get()), **CM**(A$2.get()), **CM**(A.bar()), **AM**(C.get())}
1	**LC**<A$1,C.get()>	{**AM**(C.get), **AM**(A$1.get()),**CM**(A$1.get()), **CM**(A.foo()), **AM**(A$2.get()), **CM**(A$2.get()), **CM**(A.bar()),**LC**<A$1,C.get()>}
10	**AM**(A$2.get())	{**AM**(A$2.get()), **CM**(A$2.get()),**CM**(A.bar())}

(d) Affecting Changes and To-Be-Applied Lists After Buddy and Anonymous Expansion

Figure 3.5: Affecting Changes and Intermediate Versions

explore any combinations of the affecting changes, the intermediate versions' A.bar() would become Figure 3.5(b) and A.foo() would become Figure 3.5(c). This is due to the mismatch of A$1 and A$2 between the original and edited versions. Both of these methods are now semantically inconsistent with the edited version. This can be remedied by adding **CM**(A.foo()) and **CM**(A.bar()) in the to-be-applied lists of these anonymous affecting changes.

Since locating anonymous changes within a method is tedious and prone to errors, *Crisp* ignores the application of these changes individually and instead, applies a change of the enclosing method's body. Therefore, the enclosing method's **CM** change and its transitive closure of prerequisites needs to be included in the to-be-applied list of an affecting change that is an anonymous change itself or that contains a prerequisite anonymous change. The result of anonymous expansion is shown in Figure 3.5(d). Notice that selecting any of the affecting changes 5, 6, 7, 8, or 1 essentially results in the same intermediate version, which causes the failing of testA(). During debugging (discussed in Chapter 5), anonymous expansion may lead to the identification of a change (**CM**) that encompasses a larger portion of the edit than the failure-inducing anonymous change itself.

3.4 Ordering Changes Based on Dependences

Some orderings among changes are necessary while others are undesirable. Ordering is in fact, the first step that *Crisp* takes when it receives *Chianti's* prerequisites set. Yet, it is discussed here after buddy changes because it is only after the merging of buddy changes that certain orderings of changes become problematic.

Figure 3.6 shows an addition of an abstract method A.bar(). **AM**(A.bar()) has a declaration prerequisite of **AM**(B.bar()) because without it, a compilation error will be generated due to the missing implementation of bar() in B. Due to buddy changes, **CM**(B.bar()) is added to the to-be-applied list of **AM**(A.bar()). However, **CM**(B.bar()) has also a declaration prerequisite of **AM**(A.bar()) without such a compilation error will be generated since the variable c is of type A at compilation time.

```
abstract public class A {
  public String foo() {
    ~~return ‘‘abc’’;~~
    return bar();
    });
}
  abstract public String bar();
}

public class B extends A {
  public String bar() {
    A c = new C();
    if (...)  return c.bar();
    else return ‘‘abc’’;}
}

public class C extends A {
  public String bar() {
    return ‘‘def’’;}
}
```

Figure 3.6: Ordering Changes in *Crisp*

This has now created an unnecessary dependence ordering cycle which might confuse *Crisp* when it attempts to choose which of A.bar() or B.bar() to add first. However, the absence of such an ordering does not hinder the application of A.bar() and B.bar() to class A and B respectively, as long as both of these methods are present in the final intermediate version. Hence, *Crisp* makes sure that **AM**(B.bar()) and **CM**(B.bar()) is in **AM**(A.bar())'s to-be-applied list but that their order is arbitrary.

Structural dependence is the only category of dependence that *Crisp* uses to order between a prerequisite and its dependent. Adding a class before its member fields and methods is a typical use of structural dependence. Another example is changing the type of a field that requires the field in the original program be deleted, before adding the field from the edited program. This ensures that there are no duplicate fields in the resulting intermediate versions.

Chapter 4

Interface of *Crisp*

In the previous chapter, *Crisp* focuses on the completeness of the prerequisites in the to-be-applied lists. After the expansion process, the to-be-applied lists may grow significantly in size. The intuitive effect of expansion is that the number of intermediate versions are reduced to only those that are compilable. Once expanded, *Crisp* presents these to-be-applied lists to the programmer and facilitates the exploration process in a graphical user interface (GUI). The usability of this interface and the efficiency of applying changes and creating intermediate versions are essential in the design of *Crisp* as an interactive tool.

4.1 Reduction of Affecting Changes and To-Be-Applied Lists

Once the to-be-applied lists are expanded with all the necessary prerequisites to create intermediate versions, *Crisp* goes through a process to eliminate redundant prerequisites. This process is deemed necessary especially in the the interactive mode to avoid confusing the user. It is also used in the other modes to increase the efficiency of *Crisp*. Unlike prerequisite expansion which is necessary to ensure compilability, this reduction of redundant changes is optional and is coupled with the presentation of the affecting changes to the user (discussed in Section 4.3). Since the affecting changes that remain in *Crisp* after this phase have slightly different syntactic meaning from the affecting changes in *Chianti*, they are referred to as *editable changes* to avoid confusion.

4.1.1 Reduction Due to Buddy Changes

During the interactive mode in *Crisp*, the programmer is presented with a list of affecting changes and this list is sorted according to some pre-defined heuristics for fault localization. Explicitly presenting the buddy changes in this list can be confusing. First of all, buddy changes have exactly the same to-be-applied lists. Showing both of them in a GUI is therefore, redundant. Even worse, showing them together requires an understanding of their definitions in *Chianti* because they represent the same edit of the source code in *Crisp*. Hence the change categories presented by *Chianti* go through a process to merge the buddy changes into editable changes during this phase of *Crisp*:

Buddy Changes	*Editable Changes*	*Meaning*
AM-CM	**AM**	Add a method and its body
CM-DM	**DM**	Delete a method and its body
AF-CFI	**AF**	Add a field and its initialized value
CFI-DF	**DF**	delete a field and its initialized value
AF-CSFI	**AF**	Add a field and its initialized value
CSFI-DF	**DF**	Delete a field and its initialized value
AI,CI,DI, ASI,CSI,DSI	**Initializer Change**	All initializer blocks from the edited program are applied

Table 4.1: Re-definitions of Changes in *Crisp*.

Since a buddy prerequisite and its buddy dependent may both be included in *Chianti*'s affecting changes set, this phase may result in a decrease in the number of changes a programmer can select in the interactive mode. Furthermore, it may decrease the sizes of the to-be-applied lists where buddy changes are present.

4.2 Reduction Due to Mapping Dependences

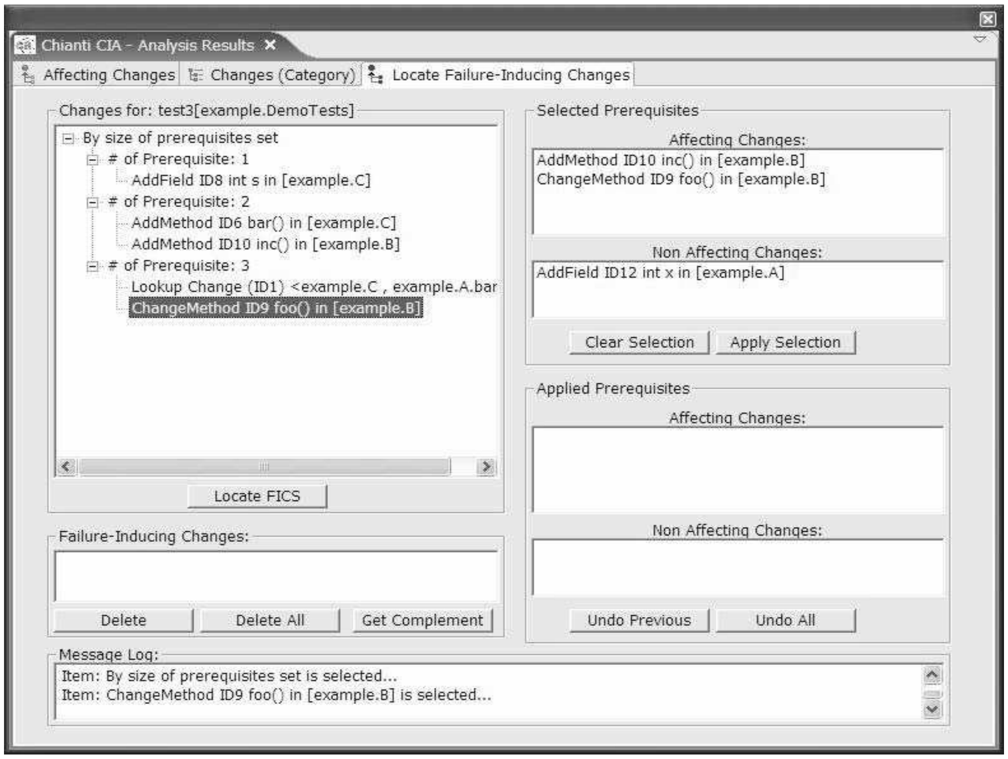
There are two major mapping dependences provided by *Chianti*. An **LC** change is the result of adding or deleting a class, or adding or deleting methods; a method change for init() or clinit() is the result of changing the implicit constructor or static initializer in a class. An **LC** change indicates a change of dispatch behavior in the program, and this information may be valuable to the programmer. However, method changes of init() and clinit() are included as affecting changes only to ensure the complete calculation of changes that involve initialization of fields. They have no visible effects in terms of the changes to the source code, nor do they provide any additional information for the programmer. Hence, *Crisp* processes all the affecting changes and their to-be-applied lists and eliminates all the method changes related to init() and clinit().

Similar to the reduction of buddy changes, this process may reduce the number of affecting changes a programmer is allowed to select, but cannot hinder the finding of faults in the program. However, depending on the type of reductions used, the number of affecting changes, and hence failure-inducing changes, reported in this thesis may be different from previous publications [5] and [12]. This will be discussed further in Chapter 6. The default implementation of *Crisp*'s interactive mode uses both buddy and mapping reductions and presents the user with the most concise sets of editable changes and their to-be-applied lists.

4.3 Presentation of Affecting Changes

Upon determining for each affecting change its to-be-applied list, the entire list of affecting changes (or editable changes) will be presented to a programmer in a GUI like the one shown in (Figure 4.1). The size of this list depends on the original number of affecting changes calculated by *Chianti* as well as the type of reductions used during the processing of the affecting changes as discussed in Section 4.1. Currently, two ways of ordering the affecting changes have been implemented in *Crisp*: (1) random and (2) by the size of their to-be-applied lists in ascending order. In Figure 4.1, the ordering shown is by the size of the to-be-applied lists in ascending order. Both of these orderings are created with reduction of buddies and mapping changes. Assume that the programmer chooses to explore intermediate version creation manually. From the ordered list on the left hand side of the GUI, the programmer can select an affecting/editable change and its to-be-applied list will be shown in the "Selected Prerequisites" lists. Since a to-be-applied list can contain both affecting and non-affecting changes, they are distinguished. Once the selected prerequisites lists are non-empty, the programmer can initiate the application of changes to the current source code of the program by clicking the "Apply Selection" button. Upon successful application of changes, the applied changes will be shown in the "Applied Prerequisites" lists.

Crisp keeps track of all the changes that have been applied in the source code. The



```
By size of prerequisites set
  # of Prerequisite: 1
    AddField ID8 int s in [example.C]
  # of Prerequisite: 2
    AddMethod ID6 bar() in [example.C]
    AddMethod ID10 inc() in [example.B]
  # of Prerequisite: 3
    Lookup Change (ID1) <example.C , example.A.bar
    ChangeMethod ID9 foo() in [example.B]
```

Selected Prerequisites — Affecting Changes:

```
AddMethod ID10 inc() in [example.B]
ChangeMethod ID9 foo() in [example.B]
```

Non Affecting Changes:

```
AddField ID12 int x in [example.A]
```

Message Log:

```
Item: By size of prerequisites set is selected...
Item: ChangeMethod ID9 foo() in [example.B] is selected...
```

Figure 4.1: A Screenshot of *Crisp* Graphical User Interface

programmer has the option to undo the previous application of changes by clicking the "Undo Previous" button, or undo all the applied changes and restore the original program by clicking the "Undo All" button. Each change is applied once and only once. Since affecting changes may share prerequisites, only the changes that have not been applied are shown in the "Selected Prerequisites" lists.

Any changes listed under the "Affecting Changes" lists can be dragged and dropped into the "Failure-Inducing Changes" list. The programmer can gather all the suspicious failure-inducing changes in this list, restore the original program by clicking the "Undo All" button, and then click the "Get Complement" button to request *Crisp* to calculate the complement set of the corresponding failure-inducing changes. The complement set will then be listed under the "Selected Prerequisites" lists. The programmer now has the option to apply the complement set to the original program. The complement set can be used to confirm whether the programmer has identified all the failure-inducing changes. A detailed discussion of complement set can be found in Section 5.3 of the next chapter.

Immediately under the ordered list on the left hand side of the GUI is the "Locate FICS" button. Selecting this button will launch the automatic mode of Crisp. A detailed discussion of the automatic algorithm for locating failure-inducing changes can be found in Section 5.5.

4.4 Intermediate Program Versions

Although dependence is critical in the creation of semantically and syntactically sound intermediate program versions, there are circumstances where *Crisp* relies on the Eclipse environment to provide additional assistance when applying changes. First and foremost, the Eclipse Java Development Tool (jdt) provides abstract syntax tree information and source code manipulation capability which allow *Crisp* to accurately pinpoint the position of changes within a piece of source code. Overall, *Crisp* makes four types of changes:

- Class changes

- Initializer block changes
- Field changes
- Method changes

To manipulate source code changes, *Crisp* first identifies the compilation units (.java files) that are involved in the changes. Only **AC** changes are allowed to be applied without an existing compilation unit. Before changes are applied to a compilation unit, the source code is cached such that any subsequent exception results in *Crisp* restoring a previously created valid intermediate version. After *Crisp* applies an affecting change and its to-be-applied list, it invokes a build of the program within Eclipse. During the build, `import` statements are organized if necessary. This phase of *Crisp* may result in a prompt for input from the user when Eclipse cannot resolve a type that has the same name as other classes declared in different packages.

Crisp only makes changes to the program that *Chianti* considers as the original program. The edited program is left untouched structurally. *Chianti* and *Crisp* require both the original and the edited programs to exist in the same Eclipse *workspace* in order to perform the analysis. Since these versions are essentially the same program with the same name within Eclipse, users who use the tools need to rename the two different versions such that they appear to be different programs in the Eclipse environment.

4.4.1 Class Changes

Java requires that the name of the class and the name of the compilation unit where its source code is located be the same. However, there are also inner classes that reside within another class, as well as special *non-public* classes that share a compilation unit with another class. Inner classes are added at the end of their enclosing class. Special non-public classes are appended to the end of the compilation unit in which they appear. *Crisp* checks to make sure that these classes are added without creating unnecessary compilation units.

To add a regular new class, *Crisp* creates a new compilation unit and then adds the class declaration with an empty class body. Deleting a regular class also results in deleting the compilation unit from the underlying file system.

4.4.2 Initializer Block Changes

For any initializer block changes in a class, all the initializer blocks from that class in the original program are deleted and those from the edited program are grouped together and are appended to the end of the class. This avoids dealing with individual blocks that may have different relative positions (hence different numerical names) in the original program and the edited program.

4.4.3 Field Changes

All fields are added to the end of their enclosing class. This is to prevent compilation errors when a field i's initializer depends on another field j, while j is located after i in the source code. For **CFI** or **CSFI** changes, the position of the field does not change in its enclosing class.

4.4.4 Method Changes

Method changes are handled the same way as field changes. All new methods are appended to the end of their enclosing class. Changed methods stay in their same position. A method that changes its return type results in *Chianti* generating a **DM** and then an **AM** connected by a structural dependency. *Crisp* first deletes the method from its current position and adds the method to the end of its enclosing class.

Chapter 5

Locating Failure-Inducing Changes in Crisp

5.1 A Basic Definition of Failure-Inducing Changes

A failing test (T) is a test that passes when executed against an original version of a program (P_O) and fails when executed against an edited version of the same program. Let F_T be the set of failure-inducing affecting changes with respect to T, and let A_T be T's set of affecting changes. Let P_F be the intermediate version of the program formed when F_T is applied to P_0, and P_C be the intermediate version formed when the complement set $C_T = A_T$-F_T is applied to P_0, denoted by:

$$P_F = P_0 \oplus F_T$$

$$P_C = P_0 \oplus C_T \tag{5.1}$$

F_T contains *all* the failure-inducing affecting changes, if T fails executing against P_F, yet passes on P_C. F_T is *minimal* when:

$$\sim \exists S \subset F_T, \text{ where}$$

$$P_{F-S} = P_0 \oplus (F_T - S) \text{ and}$$

$$P_{C+S} = P_0 \oplus (C_T + S) \tag{5.2}$$

such that:

1. T fails in P_{F-S}, and

2. T passes in P_{C+S}

5.2 Revised Definition of Failure-Inducing Changes

The basic definition 5.1 of failure-inducing changes in Section 5.1 is applicable only when there are no compilation errors in P_T and P_C. To apply these definitions in the context of *Crisp*, the to-be-applied list of F_T should be considered. When F_T contains multiple affecting changes, the to-be-applied list of F_T , or $TBA(F_T)$, is the union of all the to-be-applied lists of the affecting changes. For the basic definition 5.1 to create a valid intermediate version when F_T is applied to the original program, $TBA(F_T)$ must equal F_T. As explained in Chapter 3, the fact that most affecting changes have dependences on other changes, including non-affecting changes, in their to-be-applied lists complicates how failure-inducing changes should be defined in *Crisp*. In addition, finding C_T does not depend only on A_T and F_T, but also on the dependences between F_T and $TBA(C_T)$. Here is a simple example to illustrate that obtaining a valid C_T is not trivial. Let $A_T = \{A_1, A_2\}$ and A_1 is failure-inducing, or $A_1 \in F_T$. Using definition 5.1, $A_2 \in C_T$. Assume that $A_1 \prec A_2$, hence $A_1 \in TBA(A_2)$. This means that F_T and $TBA(C_T)$ are not mutually exclusive. Now, executing test T against P_C will also fail due to the presence of A_1. In this situation, creating P_C is redundant as it does not provide additional information regarding A_2. Since isolating A_2 from A_1 is deemed impossible in *Crisp*'s approach, A_2 should be taken out of C_T and inserted into F_T. Technically, this means F_T may contain some changes that are not failure-inducing, but are strongly related to failure-inducing changes. In this way, the F_T built by *Crisp* may be larger than minimal, but will allow the building of a valid C_T crucial to the exploration process. This is explained further in Section 5.5

Hence, a more practical definition of F_T with valid intermediate versions in mind would be:

$$P_F = P_0 \oplus TBA(F_T)$$

$$P_C = P_0 \oplus TBA(C_T)$$

where

$$F_T \cap TBA(C_T) = \emptyset \tag{5.3}$$

5.3 Complement Set

As shown in Sections 5.1 and 5.2, the use of complement set is critical in ensuring the completeness of the failure-inducing changes set. However, when the number of affecting changes are more than a few, manually composing the complement set becomes non-trivial. This process also is complicated by the inter-dependences among affecting changes. To further assist programmers in locating failure causes, *Crisp* automatically generates the complement set based on a given F_T.

By definition, F_T and C_T are mutually exclusive. In definition 5.1, C_T is readily obtainable given an F_T. However, problems arise when this definition is applied to form valid intermediate versions on which to execute test T. As stated in the revised definition 5.3, F_T and $TBA(C_T)$ need to be disjoint.

To ensure that *Crisp* finds a meaningful complement set, it has to take into account the dependences among the affecting changes. Algorithm 1 shows how *Crisp* calculates the complement set c_T given f_T, a subset of F_T. Note that line 3 calculates the to-be-applied list for c_T, but not for f_T. This algorithm not only returns a complement set c_T, but also changes, or expands, f_T if necessary. If there are changes in f_T which are prerequisites of changes in c_T, then the corresponding c_T changes must be removed and added to f_T.

```
1  getComplementSet($f_T$) {
2  $c_T = A_T - f_T$;
3  if $TBA(c_T) \cap f_T \neq \emptyset$ then            /* test for mutual exclusiveness */
4      for $\forall ac_i \in f_T$ do
5          if $ac_i \prec ac_j$ then
6              $f_T = f_T \cup ac_j$;
7              $c_T = c_T - ac_j$;
8          end
9      end
10 end
11 return $c_T$;
12 }
```

Algorithm 1: Algorithm for calculating the complement set of f_T

Fortunately, certain expansions of f_T due to the inter-dependence between f_T and $TBA(c_T)$ are merely a matter of completeness. For example, if $A_1 \prec_{Buddy} A_2$ and

$A_1 \in f_T$, then A_2 should be added to f_T also as *Crisp* considers buddy changes to be inseparable. Moving A_2 from c_T to f_T simply follows the approach set forth by prerequisite expansion (Section 3.3). If a buddy reduction is applied to the affecting changes, A_1 and A_2 are reduced to A_1 and A_2 is not visible to the users. In this case, having A_1 in f_T already implies that A_2 is failure-inducing. Similarly, if f_T contains another affecting change A_3 and $A_3 \prec_{Mapping} A_4$, having A_3 as failure-inducing implies that A_4 also is failure-inducing. Again, if a mapping reduction has been applied to the affecting changes, then A_4 will not be visible to the users; the failure-inducing nature of A_4 is implied through A_3. In practice, an F_T generated by definition 5.3 should not be much larger than the one created from the basic definition 5.1 unless the dependence graph for the affecting changes is very complex.

5.4 Types of Failure-Inducing Changes

Using definition 5.3, it is assumed that *Crisp* is capable of locating multiple failure-inducing changes. Assume that test T passes on the original program and fails on the edited program. Then, there are two major categories of the failure-inducing changes sets:

1. F_T contains only one affecting change. When this change is applied to the original program, test T fails on the intermediate version. Also, $TBA(C_T)$ does not intersect with F_T. This single affecting change in F_T is called an *explicit* failure-inducing change.

2. F_T contains multiple affecting changes. There are three types of affecting changes that constitute a multiple failure-inducing changes set:

   - *Explicit*: a failure-inducing change that by itself, when applied to the original program, results in test T failing in the intermediate version. However, none of the other changes in its to-be-applied list are failure-inducing.

   - *Implicit*: a failure-inducing change that by itself, when applied to the original program, does not result in test T failing; however, this change interacts with other failure-inducing changes in F_T and in combination they cause test T to fail. None of the other changes in its to-be-applied list are considered failure-inducing.

- *Dependent*: a failure-inducing change that is dependent on another failure-inducing change; that is, its to-be-applied list contains a failure-inducing change. This change is added to F_T by Algorithm 1 in order to generate a valid complement set.

5.5 Automatic Algorithm

Given the complement sets, a programmer now has a mechanism to confirm whether or not he has uncovered all the failure-inducing changes for a failing test. This eliminates the need to explore all combinations of affecting changes when the number of failure-inducing changes is not known in advance. However, manually determining and recording which affecting changes belong to F_T, re-executing the test after each application of changes, and requesting complement sets from *Crisp* is still a tedious process for a programmer when the failing tests contain numerous affecting changes. This process is further complicated by multiple failure-inducing changes that exhibit different behaviors as discussed in Section 5.4. *Crisp*, therefore, provides an automatic algorithm that attempts to locate failure-inducing changes with minimal programmer input. This automatic algorithm terminates when a F_T is found such that (i) T fails in $P_0 \oplus TBA(F_T)$ and passes in $P_0 \oplus TBA(C_T)$ and (ii) F_T and $TBA(C_T)$ are disjoint.

From Algorithm 1, it is clear that F_T could become large after expansion to include all the dependent affecting changes. In the worst case, $F_T = A_T$ and the programmer has to fall back to the semi-automatic interactive process to find failure-inducing changes.

Although *Crisp* does not guarantee to find a F_T smaller than A_T, it does make an effort to do so. The key to achieving this goal is to minimize the size of the set of changes that needs to be applied between two consecutive test executions. Ideally, one affecting change should be applied at a time - when a test passes before the application and fails afterwards, *Crisp* can easily pinpoint the culprit. Ordering the affecting changes based on the dependence graph could be time-consuming given the complexity of the graph with its different types of edges. As discussed in Section 3.4, ordering affecting changes can result in cycles due to buddy expansion. Instead of using dependences directly, *Crisp* orders the affecting changes by the *size of their to-be-applied lists*. An affecting change with a large to-be-applied list should be applied to form an intermediate version only after first applying those without any prerequisites. *Crisp* arbitrary selects an affecting change to break a tie. Once *Crisp* has an order for applying the affecting changes, the reversal of this order determines the next affecting change to undo when necessary. In the implementation of the algorithm, *Crisp* uses this reverse order to control the undo operation.

The control of the automatic algorithm is based on the outcomes of test results on certain valid intermediate versions. When a single failure-inducing change f (and its prerequisites) is applied to the original program and the test fails in the resulting intermediate version, f can be either an explicit or a dependent failure-inducing change. If the complement set does not fail, then f is an explicit failure-inducing change and there are no more failure-inducing changes among the remaining affecting changes. If the complement set also fails, then the algorithm needs to continue the exploration process to find additional failure-inducing changes. The type of f cannot be determined at this point. Notice that if f is a dependent failure-inducing change, the algorithm will find the failure-inducing prerequisite of f. This is because the complement set of f includes the prerequisites of f.

Crisp needs to handle implicit failure-inducing changes differently. Assume that intermediate version P_1 is formed by applying a_T, a subset of A_T, and P_2 is formed by applying a_T and f. Also assume that the test passes in P_1 but fails in P_2. When this happens, we need to examine f alone by applying f to the original program to form P_f. If the test also fails in P_f, then we can follow the aforementioned procedure for checking the complement of a single failure-inducing change. However, if the test passes in P_f, then f is an implicit failure-inducing change, meaning that there are other changes in a_T that f needs to interact with in order to exhibit failure-inducing behavior. The algorithm then needs to make sure that f is present in all future intermediate versions explored, and then all the affecting changes in a_T have to be re-explored in the presence of f. In order to make sure that f is in all future intermediate versions, the order that was originally established based on the sizes of the to-be-applied lists has to be altered and f should be inserted as first in the ordering. Since implicit failure-inducing changes require special handling, *Crisp* keeps track of them separately. In addition, *Crisp* cannot use the complement of f or the complement of $(a_T + f)$ to confirm the completeness of the implicit failure-inducing set. This step of validating with the complement set has to be deferred until *Crisp* locates all the implicit failure-inducing changes.

```
Input: $O$: an ordered list of affecting changes based on the size of their
       to-be-applied lists
1 $P' \leftarrow P_0$;
2 while there are still affecting changes in $O$ to apply do
3     $ac$ = next affecting change to apply ;
4     $P' = P' \oplus \triangle TBA(ac)$    /* apply changes in $ac$'s to-be-applied list
      that have not been previously applied */
5 end
```
Algorithm 2: *apply*(O)

In the automatic algorithm, *Crisp* has to to keep track of the intermediate versions that are generated, as well as the order in which affecting changes are applied. Here are a few $functions$ and *notations* that are used in the algorithm, defined as Algorithm 3:

1. O: An ordered list of affecting changes. This list may be re-ordered as the algorithm finds implicit failure-inducing changes.

2. $reverse(O)$: Returns a reversed ordered list O_r, based on O. In practice, *Crisp* explores changes in a reversed order of O. This is explained further below.

3. *apply*(O): This function, shown in Algorithm 2, applies all the affecting changes to the original program according to their ordering in O. Basically, each application of an affecting change results in *Crisp* applying the prerequisites in the affecting change's to-be-applied list that have not been applied already. (Recall a change is its own prerequisite)

4. $P' \ominus ac$: Affecting change ac is being *removed* from the intermediate version P'.

Since *Crisp* keeps track of $\triangle TBA(ac)$ when applying an ac in Algorithm 2, it uses the same $\triangle TBA(ac)$ to remove ac from P'.

5. f_T: This set keeps track of all the failure-inducing changes, and it grows to become F_T.

6. $f_{Implicit}$: This set keeps track of the implicit failure-inducing changes.

Algorithm 3 outlines the details of the algorithm. Here is an overview of it:

- By integrating with *Chianti*, Crisp can be invoked for any affected tests, not just the failing tests. Also, there are instances where the failure of a test is caused by changes other than the source code edit. Hence, the first step in the automatic algorithm (lines 4 - 5) is to confirm that by applying all the affecting changes to the original program in fact, the test fails. If not, the algorithm will terminate.

- Once the changes have been applied, it is more efficient to simply undo them while locating the failure-inducing changes (lines 7 - 9). In the automatic algorithm, *Crisp* applies and keeps track of the changes according to the order in O such that it can also undo the changes based on O_r, the reverse ordering of O. An affecting change is considered failure-inducing when the test result changes from failing to passing (lines 10 - 11).

- The algorithm identifies the difference between an explicit or an implicit failure-inducing changes in lines 13 - 14, by executing the test against an intermediate version where the current failure-inducing change is applied to the original version.

- Explicit failure-inducing changes are handled in lines 14 - 21. The algorithm attempts to confirm the completeness of f_T by using its complement set (line 16 - 17). When the complement set does not fail the test, the full failure-inducing changes set has been found, the algorithm exits and returns f_T (line 18). Otherwise, the algorithm continues to undo changes from an intermediate version P' that is free from any explicit failure-inducing changes (line 20).

```
Input: $O$ = ordered list of all affecting changes (AC) from the smallest
       to-be-applied list to the largest
$f_T = \emptyset$;
$f_{Implicit} = \emptyset$;
$P' = P_0$; /* $T$ runs on $P'$ */
apply($O$); /* $P'$ is now the edited program with all changes */
if $T$ passes then return;
$O_r \leftarrow$ reverse($O$);
while $f_T \neq A_T$ do
    $ac \leftarrow$ next AC to explore;
    $P' = P' \ominus ac$ ;
    if $T$ passes then
        /* found a failure-inducing change */
        $f_T = f_T \cup ac$ ;
        $P' = P_0 \oplus TBA(ac)$ ;
        if $T$ fails then
            /* failure-inducing change is explicit */
            $c_T \leftarrow$ getComplementSet($f_T$) ;
            $P' = P_0 \oplus TBA(c_T)$ ;
            if $T$ passes then  return $f_T$;
            else
                /* continue exploration */
            end
        else
            /* failure-inducing change is implicit */
            $f_{Implicit} = f_{Implicit} \cup ac$ ;
            $P' = P_0 \oplus TBA(f_{Implicit})$ ;
            if $T$ fails then
                /* found a group of implicit failure-inducing changes */
                /* that causes $T$ to fail */
                $c_T \leftarrow$ getComplementSet($f_T$) ;
                $P' = P_0 \oplus TBA(c_T)$ ;
                if $T$ passes then  return $f_T$;
            end
            else
                /* continue exploration */
            end
        end
    end
end
```

Algorithm 3: Automatic Algorithm for Locating the Failure-Inducing Changes Set for Test T

- Implicit failure-inducing changes are handled in lines 23 - 34; they are added to $f_{Implicit}$ (line 24) in addition to f_T. Lines 25 - 26 attempt to test for the completeness of $f_{Implicit}$. An implicit failure-inducing change by itself does not fail a test, but a complete set of implicit failure-inducing changes does. When this happens, the algorithm calculates the complement set (lines 28 - 30) to determine whether or not to continue the exploration. Like line 20, line 33 allows the algorithm to continue the exploration from an intermediate version that is free from any explicit failure-inducing changes. However, all the implicit failure-inducing changes are applied to the future intermediate versions in order to help identify other implicit failure-inducing changes.

This algorithm eventually terminates since the sizes of f_T and O are limited by the size of A_T. Each occurrence of a test passing results in f_T, $f_{Implicit}$, or both, expanding their sizes. Each iteration over the while loop in line 7 therefore, decreases the set of remaining affecting changes to explore. Because of implicit and multiple failure-inducing changes, *Crisp* needs to alter O based on the changes in f_T and $f_{Implicit}$. Hence the intermediate version needs to be reset in line 20 and line 33 before the algorithm continues to explore changes.

5.6 Interactively Locating Failure-Inducing Changes

Since the automatic algorithm does not guarantee finding a minimal set of failure-inducing changes, the programmer may need to manually explore the effects of applying different affecting changes to the original program. As mentioned before, when the size of the affecting changes set is not small and the software is complex, manually applying and undoing each of these changes can be a tedious task. Therefore, the use of a heuristic to rank the likelihood of an affecting change being failure-inducing may be beneficial.

Crisp comes with two standard built-in heuristics. Additional heuristics can be added by extending the `AbstractGrouping` class and implementing the `public void order()` method. The effectiveness of the two standard built-in heuristics will be discussed in Section 6.3. The most basic heuristic is *random* in that a random generator is used to assign ranking to each of the affecting changes of a failing test. These affecting changes are then ordered in a “Tree” display format for programmer selection. The second heuristic is ranking by the size of the to-be-applied set. This heuristic is also used in the automatic algorithm to generate the next smallest number of affecting changes to apply to the original program.

Chapter 6

Experimental Results

The advantages of *Crisp* can only be realized in the development of medium to large size source code programs where test failure occurs due to more than a handful of affecting changes. Furthermore, since *Chianti* and *Crisp* are both designed to work within the Eclipse IDE, these programs need to include a set of tests that can exercise the code base in Eclipse.

`Daikon`, a dynamic invariant detector program developed by Michael Ernst at MIT, as well as the `Eclipse jdt.core` plug-in program, have been identified as open source Java projects where both the development history and test environment setup fit the required assumptions of *Chianti* and *Crisp*. Each of these programs provides more than one year's worth of change history with identifiable versions and corresponding tests that are adequate for analysis. For `Daikon`, each version pair contains a week's worth of changes during the year 2002. For `Eclipse jdt.core`, nightly builds are extracted for the years 2003 and 2004.

As mentioned in previous publications [5] and [12], failing tests are rarely found in the versions that are checked into the public repositories. To induce $artificial$ failure, tests in version n are applied to version $n + 1$. `Daikon` has been extensively analyzed during the development of *Chianti* [13]. Unfortunately, it turns out that only 1 out of more than 50 versions of `Daikon` in 2002 contains two artificial failing tests. The `Eclipse jdt.core` plug-in, on the other hand, contains some failing tests directly in the repository; to further increase the number of failing tests, artificial failing tests are also used. In the experiments described in this chapter, failing tests can be either artificial or non-artificial, except when stated specifically. Within `Eclipse jdt.core`, over 20 version pairs have been identified that contain failing tests. Table 6.1 shows the sizes of these programs in terms of their lines of code and number of Java elements. These numbers do not include any libraries used. For `Daikon`, the numbers shown are the averages of their Nov 11 and Nov 19 versions. For `Eclipse jdt.core`, the first number in each cell represent the size of the earliest version we used in our experiments for that year and the second number represents the size of the last version we used in our experiments for the same year. This provides a general idea of the range of sizes and the growth of `Eclipse jdt.core` in 2003 and 2004. The KLOC column is the number of thousands of lines of non-blank and uncommented code. These statistics are collected using the Metrics [1] plugin of Eclipse.

Program	**Year**	**KLOC**	**# of Classes**	**# of Methods**
`Daikon`	2002	72	650	6,100
`Eclipse jdt.core` plug-in	2003	148−159	849−908	9,958−10,604
`Eclipse jdt.core` plug-in	2004	166−206	934−1,080	11,002−13,772

Table 6.1: Sizes of Experimental Data

All the tests used in the experiments are JUnit tests that come with `Daikon` and `Eclipse jdt.core`. The organization of tests within the JUnit framework requires the grouping of related test methods into JUnit *TestCases*, which further can be grouped into *TestSuites*. The granularity of analysis performed by *Chianti* is at the test method level. An affected test is therefore, a test method within a JUnit *TestCase*. For non-JUnit tests, the granularity is the `main()` method of a test class. Since *Crisp* is an extension to *Chianti*, *Crisp* calculates all the affecting changes, prerequisites, and to-be-applied lists on a per test method basis. Therefore, all the experiments described in this chapter are performed on a per test method basis.

It is not surprising that the sets of affecting changes for some test methods within the same *TestCase* are similar. It is also not surprising that test methods with similar sets of affecting changes share the same failure-inducing changes set (i.e., a *FICS*). For test methods where the affecting changes and FICS are the same, a representative test method is chosen to be the failing test for analysis. However, two failing test methods

[1] http://metrics.sourceforge.net/

are considered different failing tests when they have different FICS, even though they may have the same affecting changes.

`Eclipse jdt.core` comes with seven test packages. Only two of these packages, `tests.compiler.parser` and `tests.compiler.regression`, are used in the experiments as they have a long history of development with the `Eclipse jdt.core` plug-in and they are written to be executed as JUnit tests instead of JUnit plugin tests, which are created specifically for Eclipse plug-ins. Other failing tests that are excluded from the experiments either have fewer than three affecting changes or have test failure caused by changes other than a source code edit. Table 6.2 shows the version pairs that contain failing tests and the representative test method used in the experiments. The column "$\# AC$" indicates the number of affecting changes obtained from *Chianti*. A total of 31 failing tests have been identified and analysed in our experiments: two from the `daikon.test.diff` package of `Daikon` 2002, eleven from the `tests.compiler.parser` (or `parser`) package of `Eclipse jdt.core`, and eighteen from `tests.compiler.regression` (or `regression`) of `Eclipse jdt.core`. Five out of eleven failing tests in the `parser` package have over 140 affecting changes. Six out of eighteen failing tests in the `regression` package have twenty or more affecting changes.

It is clear from Table 6.2 that the number of affecting changes can be too high for easy manual exploration. From the results of the experiments, there is evidence that the affecting changes set alone is not adequate for constructing valid intermediate versions. In Figure 6.3, the original sizes of affecting changes sets from the 31 failing tests are grouped into 3 categories: small: with 3 to 11 affecting changes; medium: 13 to 74; large: over 140. Twenty-two out of these 31 failing tests need to have non-affecting changes dragged into their to-be-applied lists. On average, the number of the dragged in changes increases with the number of affecting changes, indicating an increase in syntactic dependences of affecting changes on non-affecting ones. Most of these non-affecting changes are dragged in due to buddy expansion. Anonymous expansion occurs only for 1 failing test.

All tests except seven benefited from the reductions of buddies and implicit constructors, effectively reducing the number of affecting changes to a smaller number of

Index	Package	Version Pairs	Failing Test Case with Method	# AC
1	daikon.test.diff	`Daikon` Nov 11-19	MinusVisitorTester.*testMinus()*	36
2			XorVisitorTester.*testXor()*	37
3	parser	jdt.core 2003 Jan 21-22	CompletionParserTest.*testDB_1FHSLDR()*	164
4			CompletionParserTest.*testV_1FGGUOO1()*	146
5			CompletionParserTest.*testZA_1*	142
6		jdt.core 2003 Jan 22-23	CompletionParserTestKeyword.*test0026()*	6
7		jdt.core 2003 Feb 14-15	FieldAccessCompletionTest.*testArrayAccess()*	11
8		jdt.core 2003 Jun 24-25	DietRecoveryTest.*test62()*	3
9		jdt.core 2004 Sep 8-9	ComplianceDiagnoseTest.*test0043()*	3
10		jdt.core 2004 Sep 27-28	ComplianceDiagnoseTest.*test0047()*	5
11		jdt.core 2004 Nov 19-20	ComplianceDiagnoseTest.*test0005()*	153
12			SourceElementParserTest.*test19()*	144
13		jdt.core 2004 Dec 8-9	AnnotationDietRecoveryTest.*test0005()*	4
14	regression	jdt.core 2004 Jan 13-14	JavadocTestOptions. *testInvalidTagsRefMethodErrorTagsPublic()*	74
15		jdt.core 2004 Mar 04-05	JavadocTestForClass.*test002()*	13
16		jdt.core 2004 Mar 19-20	AssignmentTest.*test002()*	5
17		jdt.core 2003 Mar 31-Apr 01	DeprecatedTest.*test1()*	43
18		jdt.core 2004 Apr 10-11	ClassFileReaderTest.*test002()*	36
19		jdt.core 2004 Apr 13-14	ClassFileReaderTest.*test002()*	27
20		jdt.core 2004 Jun 08-09	Compliance_1_3.*test77()*	13
21			Compliance_1_3.*test78()*	11
22		jdt.core 2004 Jun 09-10	Compliance_1_3.*test77()*	9
23			Compliance_1_3.*test78()*	10
24		jdt.core 2004 Jul 08-09	JavadocTestMixed.*testBug62812()*	6
25		jdt.core 2004 Jul 09-10	LookupTest.*test046()*	16
26		jdt.core 2004 Jul 21-22	TryStatementTest.*test023()*	16
27		jdt.core 2004 Aug 23-24	ClassFileReaderTest.*test048()*	8
28		jdt.core 2004 Nov 16-17	Compliance_1_3.*test089()*	13
29		jdt.core 2004 Nov 18-19	JavadocTestForMethod.*test034()*	36
30		jdt.core 2004 Nov 29-30	AssignmentTest.*test017()*	11
31		jdt.core 2004 Nov 30-Dec 01	ClassFileReaderTest.*test058()*	20

Table 6.2: Overview of Version Pairs with their Failing Tests

Category	# of Affecting Changes	# of Tests	# of Tests With Dragged In	Avg. # of Dragged In Changes
Small	3 to 11	13	7	0.7
Medium	13 to 74	13	10	7.7
Large	142 to 164	5	5	24.6

Table 6.3: Relationship of Affecting Changes and Non-Affecting Changes Dragged into their To-Be-Applied Lists

editable changes. Recall that these editable changes are the ones that are presented to the user for exploration. In Table 6.4, the average sizes of the affecting changes and editable changes sets per failing test in each category are listed. Interestingly, the percentage of reduction increases with sizes of affecting changes. Also, the majority of the reduction is due to buddies as shown in the "Due to Buddy Reduction" column. Overall, there is a 25% size reduction.

Category	**Avg. # of Affecting Changes**	**Avg. # of Editable Changes**	**Percent of Reduction**	**Due to Buddy Reduction**
Small	7.15	6.38	10.7%	90.9%
Medium	29.23	22.77	22.1%	87.9%
Large	149.80	106.80	28.7%	96.3%

Table 6.4: Summary of Affecting Changes and Editable Changes

Only two failing tests have completely independent affecting changes, where each to-be-applied list contains only one affecting change itself. The rest of the failing tests have some form of dependences among their affecting changes. In addition, there seems to be a correlation between the average size of the to-be-applied lists and the size of the affecting changes sets as shown in Table 6.5. The sizes of the to-be-applied lists reported in this table are collected after buddy and mapping reduction. The larger the sizes of the to-be-applied lists means that more editable changes are dependent upon each other. In some situations, changes can form dependence cycles which result in fewer valid compilable intermediate versions. Two out of the five tests in the "Large" category contain exceptionally large dependence cycles among their editable changes; these editable changes are prerequisites of one another, and the sizes of their to-be-applied lists are 114 and 120 respectively.

Category	**Avg TBA**	**Max Range of TBA**
Small	1.57	1 to 3
Medium	2.90	2 to 25
Large	52.54	25 to 120

Table 6.5: Summary of Sizes and Ranges of To-Be-Applied Lists

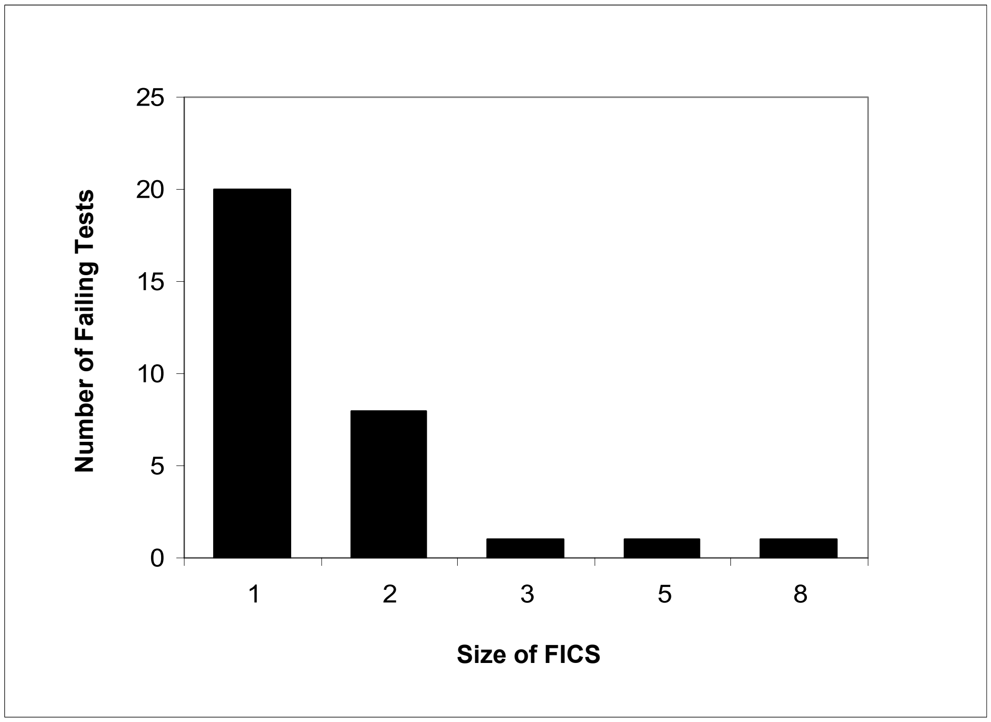
6.1 Locating the Failure-Inducing Changes Set (FICS)

The most interesting result of using *Crisp* is the fact that it identifies multiple failure-inducing changes (see Section 5.4). Out of the 31 failing tests, only 20 of them have a single explicit failure-inducing change. The remaining 11 failing tests have multiple changes in their FICS. Among them, 6 failing tests have multiple explicit failure-inducing changes, 4 have multiple implicit failure-inducing changes, and 1 contains multiple dependent failure-inducing changes. Although it is possible for failing tests to have more than one type of failure-inducing change, none of the tests in our experiments have multi-type FICS. Figure 6.1(a) shows the size of the FICS for all the failing tests.

Based on the results of the experiments, the sizes of the FICS sets do not appear to have a direct relationship with the number of the affecting changes or editable changes. This is reflected in Figure 6.1(b) in which most of the failing tests that are affected by a large number of changes have a single failure-inducing change. In Figure 6.1(c), the sizes of the FICS are listed in comparison to the average number of affecting and editable changes. Note that the FICS of size two corresponds to the smallest sets of affecting and editable changes. Neither does the average sizes of the to-be-applied lists provide an indication as to the resulting size of the FICS. Only one in 31 failing tests with multiple FICS of size 8 has an unusual high average size for its to-be-applied lists.

On average, the size of the FICS is about 5.7% of the size of editable changes sets generated. *Crisp* improves considerably over the focus provided by *Chianti* which selects the affecting changes from all the changes in an edit. These results show that *Crisp* can be a viable solution to focus programmers on the few changes that are relevant to the causes of failure.

Of all the failing tests, locating the failure-inducing changes in `CompletionParserTest.testDB_1FHSLDR()` in version pair `jdt.core 2003 Jan 21-22` was the most tedious if done semi-automatically. First of all, the number of affecting changes, (164), is the highest among all the failing tests that were identified. Second, all of the 3 changes in the FICS are implicit. Finally, all the changes come from the



(a) Distribution of Failure-Inducing Changes Set Sizes

Size of AC	Percent of Tests with Single FICS	Largest FICS set
Small	62	2
Medium	62	8
Large	80	3

(b) Affecting Changes and FICS

FICS Size	Avg. # Affecting Changes	Avg. # Editable Changes	Avg. Size of TBA	FICS as % of Editable Changes
1	43	32	44	3.1
2	16	13	2	15.3
3	164	111	4	2.7
5	27	19	1	26.3
8	43	40	7	20.0

(c) FICS, Sizes of Changes and To-Be-Applied Lists

Figure 6.1: Overview of Affecting Changes and Failure-Inducing Changes

`CompletionParser` and its parent `AssistParser` class that contains multiple affecting changes. The fact that only these 3 are failure-inducing is not obvious. Without an automatic algorithm, the only other way to locate these changes efficiently is to use heuristics and hope that all of these changes are ranked in the first few that the programmer should explore.

The other interesting outcome of the experiment comes from the failing test `DeprecatedTest.test1()` in version pair `jdt.core 2003 Mar 31-Apr 01` where there are eight failure-inducing changes out of 43 affecting changes. Thirty-five non-affecting atomic changes are dragged into the to-be-applied lists due to buddy and anonymous expansion. All of the eight failure-inducing changes are related to a change of type hierarchy (**CTD**) of an abstract class `AbstractVariableDeclaration`. Among them, seven form a cycle of dependence where each has a to-be-applied list that contains the other 6 changes as prerequisites. All these changes are considered dependent failure-inducing changes as *Crisp* is incapable of applying them separately. The eighth change is also considered a dependent failure-inducing change since its prerequisites set contains the other seven changes. In other words, these eight changes are inter-dependent failure-inducing changes. Intuitively, all syntactic changes due to a type hierarchical change have to be applied together.

6.2 Efficiency of the Automatic Algorithm

The automatic algorithm systematically locates the FICS with minimal input from the user. The efficiency of the algorithm depends mainly on six factors:

1. The execution time of the test case in JUnit;
2. The program build time after each application or undo of changes;
3. The number of editable changes;
4. The rank of the failure-inducing changes within the editable changes;
5. The size of the failure-inducing changes set;

6. The nature of the failure-inducing changes.

The execution time of a JUnit TestCase depends not only in the nature of the test methods but the number of test methods that are within it. Although *Chianti* and *Crisp* both perform analysis at a test method level, the current interface between *Crisp* and JUnit does not isolate the execution of a single test method within a TestCase. All test running times presented here are therefore, the running time of the TestCase containing the failing test method. This basically mimics the way a user would use *Crisp* in the semi-automatic interactive mode. After the user instructs *Crisp* to apply or undo a set of changes, he/she has to invoke JUnit to re-run the TestCase and observe the result of the test method(s). The average time to execute the TestCase of interest in `Daikon` is less than 1 second, in the `Eclipse jdt.core parser` package is less than 3.5 seconds, and in the `Eclipse jdt.core regression` package, 32 seconds.

The number of times a TestCase needs to be executed within the automatic algorithm depends on the ranking of the FICS within the editable changes as well as the size of the FICS set. Since the automatic algorithm begins by undoing a change from all the applied editable changes, a high ranking of the FICS in the algorithm should yield better efficiency. Hence, the quality of change rankings can significant affect the efficiency of the algorithm.

The program build time depends on the number of compilation units that have been altered from one intermediate version to the next, as well as whether `import` statements need to be added due to compilation errors caused by unknown types. This could potentially result in the algorithm prompting and waiting for user's input when a specific `import` statement is needed for a type name that is found in multiple packages of the build path. During the 31 runs of the automatic algorithm, there were three occasions when Eclipse prompted for such input. Every application and undo of changes may require two program builds: one to build after the changes are applied/undone, the other to build after organizing the `import` statements if necessary. Since the second build may not happen, the total number of builds should be between one to two times the number of test runs. In other words, the number of builds also depends on the rankings and the size of the FICS set.

For each test result that changes from fail to pass, *Crisp* has to apply the changes in the FICS set and then apply the changes in the complement set to confirm that the FICS set is complete. Hence, no matter what the rankings of the FICS within the editable changes, the number of editable changes certainly has an effect on the total execution time of the algorithm.

The relationship among the multiple changes in a FICS affects the efficiency of the algorithm in different ways. Explicit failure-inducing changes, the simplest among the three types of failure-inducing changes, cause the algorithm to continue until it finds a complement set that does not fail the test. Implicit failure-inducing changes certainly require more bookkeeping and processing time since the ordering of the editable changes needs to be revised in order to continue the exploration systematically. On the other hand, multiple dependent changes in a FICS set that resulted from expanding the FICS set to calculate a valid complement set may not affect the running time of the algorithm significantly. More dependency among changes means fewer valid intermediate versions to explore, which in turn results in fewer test executions during the algorithm.

All the experiments are performed on a machine with Pentium IV 2.8 GHz processor and 1.5G memory. The response time (from the time the user clicks the "Locate FICS" button in the user interface to the time *Crisp* reveals the FICS on the screen) of the algorithm ranges from 25 seconds to 22 minutes for our test data. The majority of this response time is spent in re-executing the TestCases and building the programs after changes are added or taken out. On average, the total time spent in executing the tests constitutes 56.7% of the response time while building the programs constitutes 19.9%. The response time of using the automatic algorithm to locate the FICS for the 31 identified failing tests is shown in Figure 6.2. It is clear that in general, total response time increases with total test running time. Since the total build time depends on the number of compilation units affected and whether or not `import` statements need to be organized, it does not appear to correlate with total response time.

When the total test run and program build time dominates the total response time, it is interesting to understand how long a single test run and program build takes for each failing test. Figure 6.3 shows the average test run time which is the total test run time

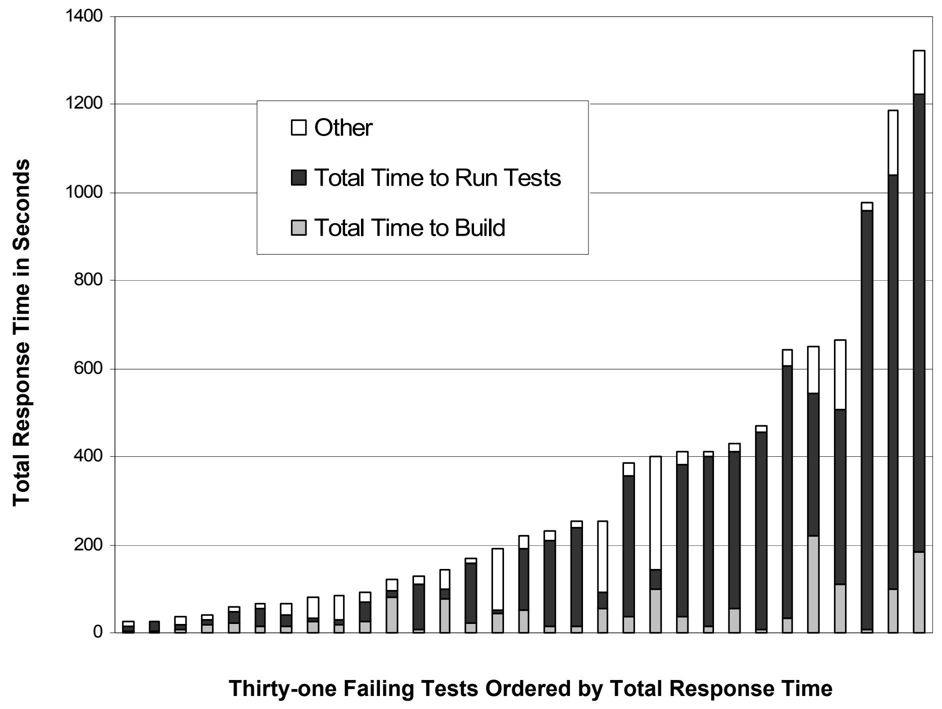


Figure 6.2: Main components of Total Response Time

divided by the number of application/undoing of changes, for each test. Similarly, the average build time is the total build time divided by the number of application/undoing of changes. The results are presented for all 31 failing tests ordered by their total response time, same as what is shown in Figure 6.2. It is clear that many tests take more than 30 seconds to run. Specifically, there are five tests that take more than 50 seconds to execute. For the failing test `Compliance_1_3.test78()` in the `jdt.core Jun08-09` Version pair which contains only eleven affecting changes, the reason why it may take more than 4 minutes to locate the FICS is that it takes almost a minute just to run the test itself! Again, the average time to build fluctuates from one failing test to the other. A spike in the average build time sometimes means that the user is being prompted for input to select specific `import` statements during the build of the program.

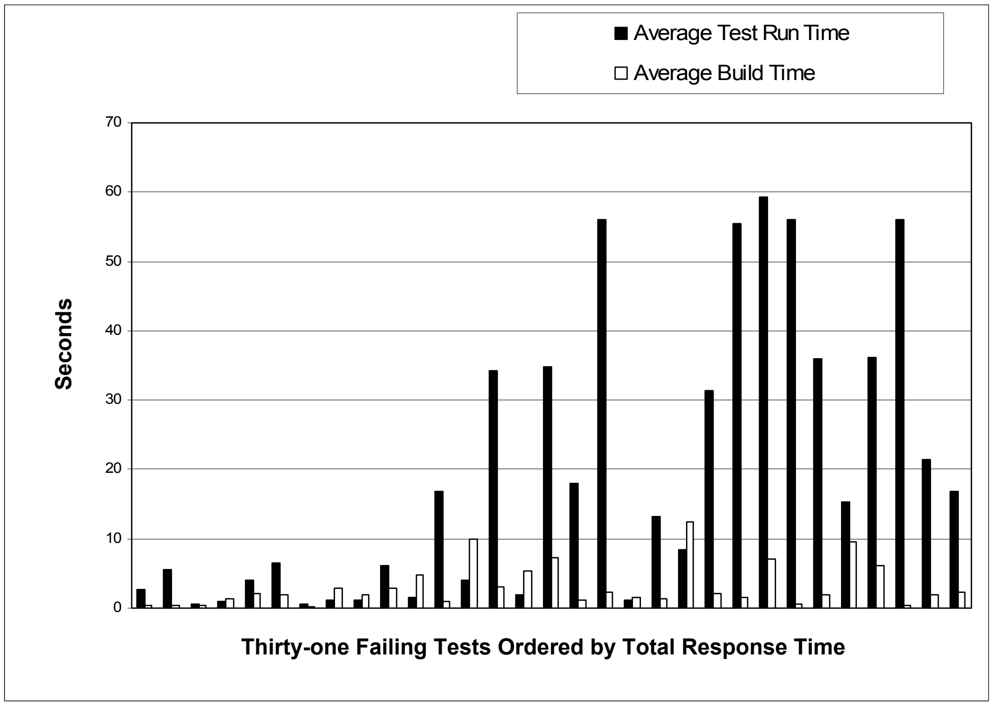


Figure 6.3: Average Test Run Time and Build Time

From Figures 6.2 and 6.3, the average test running time and average build time are under ten seconds for the first eleven failing tests with total response time under two minutes. Interesting observations can be made on the relationship between the total response time and other factors that affect the efficiency of the algorithm within this two-minute range. Figure 6.4 shows the relationship between the total response time and the size of the editable changes set multiplied by the size of the FICS. Within the two-minute range, the automatic algorithm is capable of locating the FICS for two failing tests that have almost 100 editable changes. There are also two failing tests that contain multiple failure-inducing changes: one FICS contains explicit changes and the other FICS contains implicit changes. This shows that the algorithm can be efficient for an interactive tool provided the test run time does not exceed a few seconds.

For the remaining failing tests, all of them with more than 100 editable changes have a total response time of under seven minutes. The most difficult failing tests that we mentioned earlier with 111 editable changes and a FICS that contains three implicit failure-inducing changes, was completed in 255 seconds. The failing test that contains a cycle of eight changes in its FICS with 40 editable changes was completed in 145 seconds. In these two cases, the test execution time is either exceptionally small, or the number of times the test is executed is small. Hence, the total response time can be affected by factors other than the sizes of the editable changes set and FICS.

So far the parameters discussed are determined by the nature of the program and the failing tests: the duration of test execution, the program build time, and the nature and sizes of the editable changes set and the FICS. The one parameter that *Crisp* can improve upon in the automatic algorithm is the ranking of the editable changes. This ranking affects the algorithm in two ways. First, the higher the ranking of the FICS, the sooner the algorithm detects a change of test results when it begins to undo the changes, and the fewer editable changes it needs to explore. This efficiency can easily be leveraged when there is more than one change in the FICS for the failing test. Second, the fewer the number of editable changes to explore, the fewer test runs and program builds are necessary to observe whether there is a change in the test result. When the average time for running the tests and building the program is significant, an effective

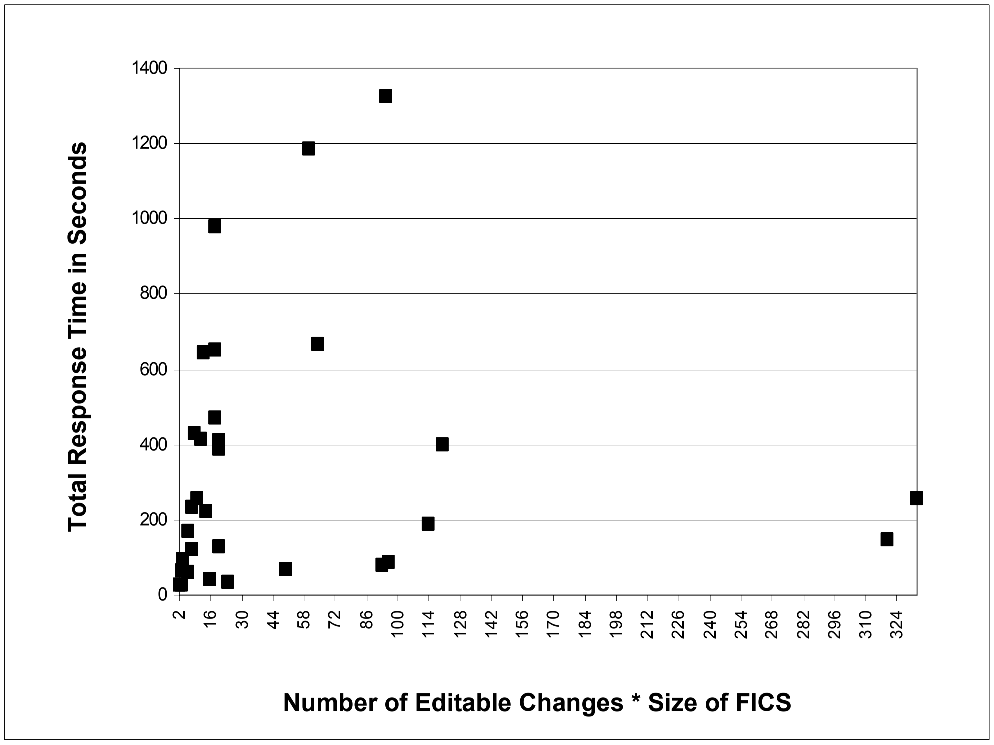


Figure 6.4: Efficiency of Algorithm in Handling Complex FICS

ranking heuristic becomes essential for an interactive tool like *Crisp*.

The default ranking used in this section is based on the sizes of the to-be-applied lists. As mentioned before, the main reason for using this ranking is the fact that it is readily available as a result of compiling the to-be-applied lists. The other reason is that it approximates the minimum number of changes that are needed for forming the intermediate version for each iteration. Having these two advantages does not necessarily mean that this type of ranking is effective. Since the total response time of the experiment ranges widely, it is interesting to understand the importance of a ranking based on the nature of the failing tests. The 31 failing tests are extracted from 3 very different test programs. `Daikon` has two failing tests for which *Crisp* is able to locate the FICS automatically in 36 and 66 seconds respectively. However, the response time per failure-inducing change does not vary significantly for these two tests. The average test run time is about 0.6 seconds. *Crisp* needs to explore 15 out of 24 editable changes to locate the single failure-inducing change in one case; 22 out of 25 in the other case where the FICS contains two changes. Even though the ranking is far less than optimal, the low average run time and build time makes the response time of the algorithm acceptable, when compared to the exploration time in a semi-automatic setting. In such a setting, the time for exploration involves having the programmer review the to-be-applied lists and select the various UI buttons for applying the changes, invoking JUnit, as well as calculating the complement sets.

In the `Eclipse jdt.core parser` and `regression` test programs, response time varies significantly. For the `parser` package, response time ranges from 25 to 399 seconds; on the `regression package`, 25 to 1,323 seconds. The expensive running time of the failing tests in the `regression` package underscores the importance of an effective ranking heuristic. The product of the number of editable changes and the size of the FICS has no direct impact on the total response time in the automatic mode (see Figure 6.4). If we multiply this product by an additional factor, the rankings of the failure-inducing changes (in terms of the percentage of editable changes explored), the resulting value referred to as *REF*, seems to have an impact on the number of times a test is executed. In Figure 6.5, the REF value is plotted against the logorithm of the number of test executions for both the `Eclipse jdt.core regression` and `parser` test packages. It is noticeable that as the REF value increases, the number of test runs increases for the `regression` test package. The only exception in the trend is when REF is around 64. This point corresponds to a failing test with eight dependent changes in its FICS. Since these eight changes were found at the same time by expanding the FICS set to calculate a valid complement set, the algorithm does not have to traverse the editable changes eight times to locate the entire FICS. Also, the system recorded that eight out of forty editable changes have been explored when in fact, only two editable changes truly have been explored. The other six are considered to be in the FICS because of the dependence cycle.

The effect of REF on the total response time is shown in Figure 6.6. As expected, the total response time also increases as the REF value increases for the `regression` test package. The exceptions are failing tests that have very few editable changes or an exceptionally small duration running time. In general, the algorithm explores fewer than 50% of the editable changes in ten out of the eighteen failing tests in the `regression` test package. However, the failing tests with the worst response time per failure-inducing change are those for which the algorithm explores more than 50% of the editable changes due to poor ranking, despite below average test running time.

In contrast, the `parser` test package exhibits different outcomes. In this package, test runs usually occur fewer than 10 times despite significant changes in the REF value. In Figure 6.5, the only obvious effect of the REF value on the number of test runs pertains to a failing test that has 111 editable changes and a FICS of size three. For these `parser` failing tests, their REF values have some, but limited effects on the total response time as shown in Figure 6.6. In fact, the total response time is dominated by the average test run time instead of the REF values for most of the failing tests in this test package. Unlike the `regression` package, the failing tests with the worst response time per FICS in this package have higher than average test execution and build times. Finally, four out of the eleven `parser` tests have fewer than 50% of their editable changes being explored. This indicates that the size of the to-be-applied list ordering heuristic is less effective for the `parser` than the `regression` tests.

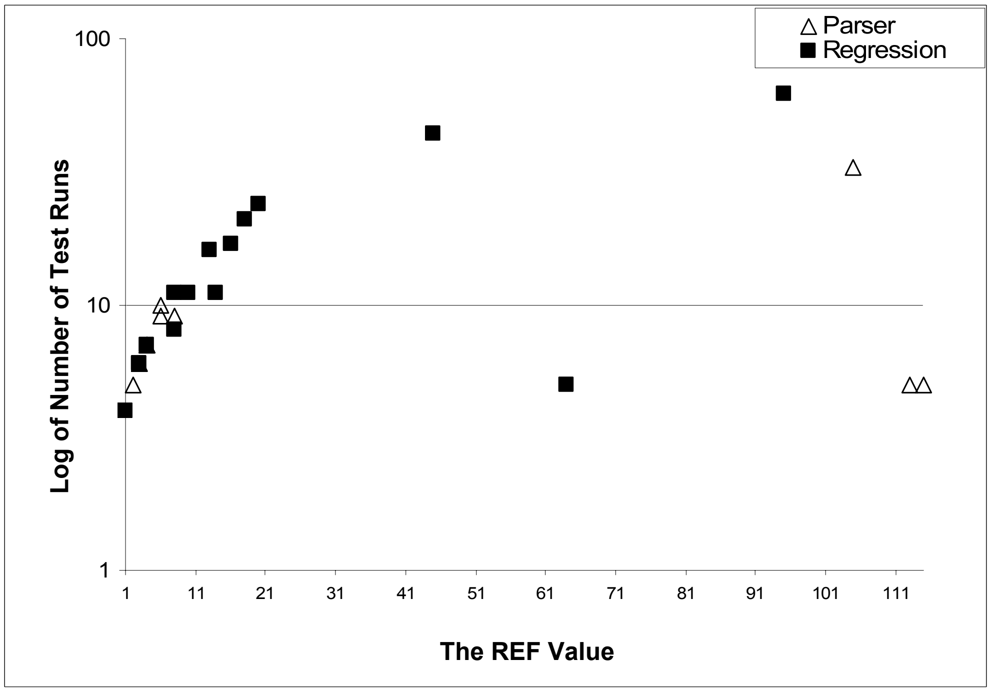


Figure 6.5: Effects of Rankings on `Eclipse` Test Package Number of Test Runs

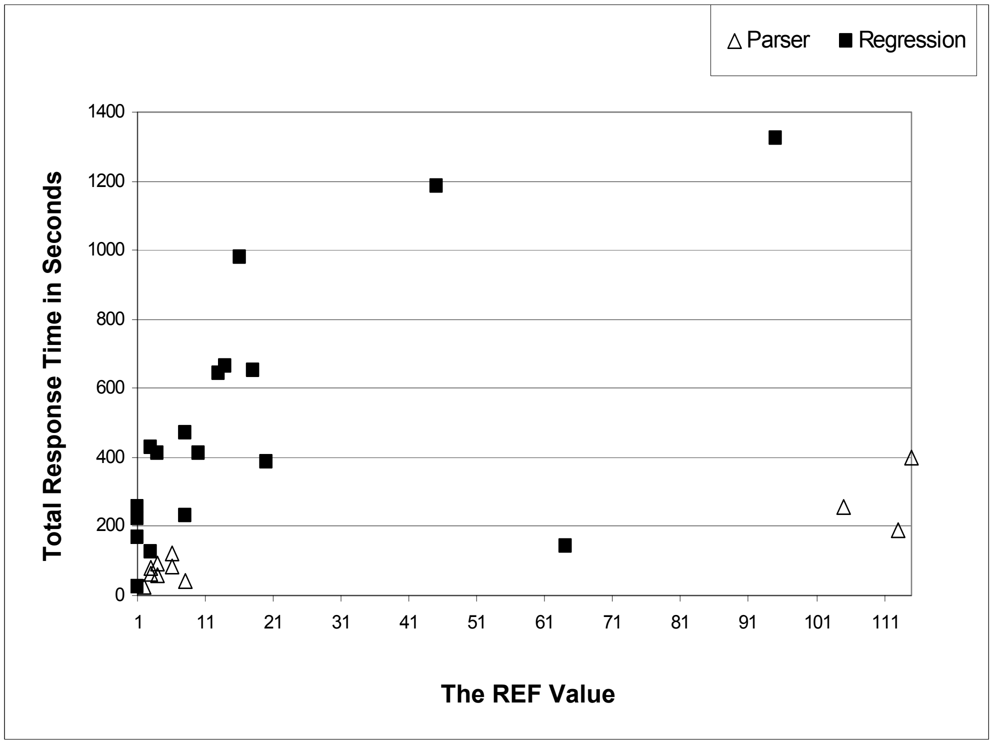


Figure 6.6: Effects of Rankings in `Eclipse` Test Package Total Response Time

In summary, using a simple ranking heuristic such as the size of the to-be-applied lists results in 14 out of 31 cases when the automatic algorithm explores fewer than 50% of the editable changes, or 8 out of 31 instances when fewer than 25% of editable changes are explored. An ineffective ranking does not necessarily result in excessive response time; it has to be coupled with other factors such as the nature of the programs, the duration of the test run and build time, the number of editable changes to explore, as well as the size of the FICS.

6.3 Comparison of Heuristics

Aside from providing an automatic algorithm to locate failure-inducing changes, users of *Crisp* can explore the impact of applying affecting changes in different orders (or rankings) in the semi-automatic mode. In this mode, the users are given different heuristics to view the editable changes. As discussed in previous chapters, the default view for the changes in *Crisp* is by the size of the to-be-applied list (i.e., the prerequisite-size heuristic). In general, addClass(**AC**) and addField(**AF**) changes have fewer prerequisites than the addMethod(**AM**) or changeMethod(**CM**) changes; therefore, **AC**s and **AF**s are listed first when *Crisp* presents the changes using this heuristic. Also implemented in *Crisp* is the *random* heuristic. Using the pseudo-random number generator provided by Java, this heuristic orders the editable changes without any consideration of their properties.

To compare the two heuristics, the automatic algorithm is re-executed for all the 31 failing tests using the random heuristic to rank the editable changes instead of the prerequisite-size heuristic. Table 6.7 shows the results of these two heuristics in terms of the number of editable changes that are explored before the automatic algorithm terminates as well as the total response time. Using the same categories used in Table 6.3, Table 6.7(a) lists the differences of the heuristics by categories. Other than the thirteen failing tests with "Small" number of affecting changes, the prerequisite-size heuristic appears to perform a little better than random. On average, it traverses a fewer number of editable changes before locating the failure-inducing changes. This

Category	Avg # of Editable Changes Explored		Avg Total Response Time (in sec)	
	Random	Prerequisite-Size	Random	Prerequisite-Size
Small	50	53	2,498	2,919
Medium	174	163	6,964	6,316
Large	321	272	1,246	1,009

(a) Comparison Based on Category

Package	Avg # of Editable Changes Explored		Avg Total Response Time (in sec)	
	Random	Prerequisite-Size	Random	Prerequisite-Size
daikon.test.diff	6	37	69	101
parser	338	292	1,707	1,412
regression	203	159	8,932	8,731

(b) Comparison Based on Different Packages/Programs

Figure 6.7: Differences Between the Random and Prerequisite-Size Heuristics

directly translated into a better total response time than the random heuristic. In Table 6.7(b), the differences are listed based on the origin of the failing tests. The random heuristic performs extremely well for the two tests in `Daikon`. However, the reverse is true for the `parser` and `regression` packages of `Eclipse jdt.core`. More interestingly, the gains of exploring many fewer editable changes (over 20% of gain) using the prerequisite-size heuristic do not translate into any significant gains in total response time (less than 2.3% of gain). Again, this could be attributed to the dominance of the test run and build time of the program. In fact, the total response time for locating the FICS for all these 31 failing tests using these two heuristics differs by only 5%, not significant to draw conclusion as to which heuristic is more efficient when used by the automatic algorithm.

Finally, Figure 6.8 shows the detailed differences of the number of changes explored between the two heuristics. The positive differences signified that the prerequisite-size heuristic performed better than random; the negative differences signified the random heuristic did better than prerequisite-size. In 15 out of 31 cases, the prerequisite-size heuristic explored fewer editable changes. The prerequisite-size heuristic is also more efficient in cases where there are multiple failure-inducing changes (the gray bars). One

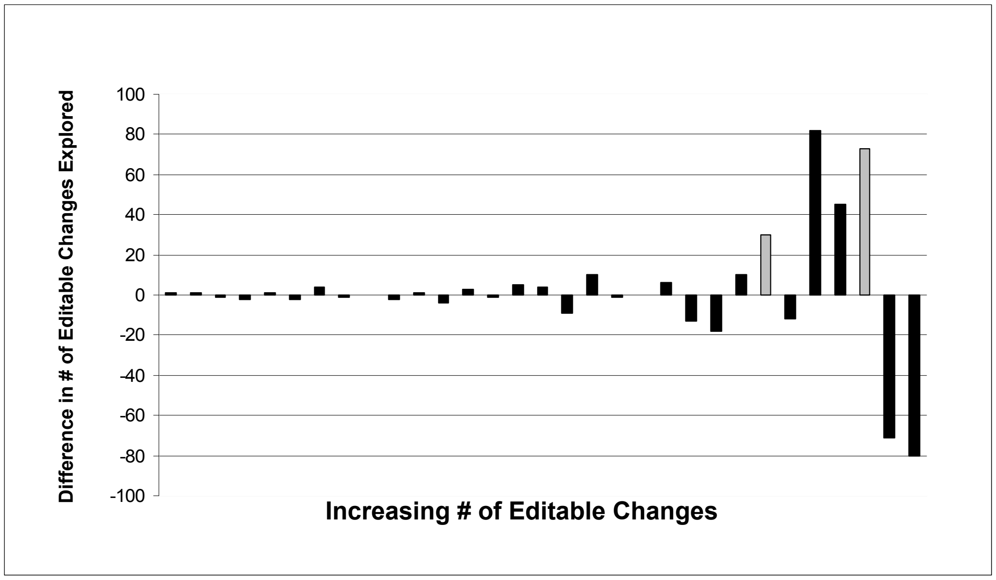


Figure 6.8: Differences Between Prerequisite-Size and Random Heuristics for 31 Failing Tests

simple explanation is that it is not likely that more than two failure-inducing changes are randomly ranked close to one another. One conclusion that can be drawn directly from Figure 6.8 is that neither the random or the prerequisite-size heuristic provides consistency in suggesting the most likely FICS to users. Especially in semi-automatic mode with a high number of editable changes, a more effective heuristic is needed to locate the FICS efficiently.

Chapter 7

Limitations of *Crisp*

Using syntactic dependences to create intermediate versions for isolating fault-inducing changes can be an effective step in the beginning of the debugging process. However, this methodology works on certain assumptions and exhibits some limitations that are discussed in the following sections.

7.1 General Assumptions for *Crisp*

The test suite that exercises the original and edited versions of a program needs to be comprehensive such that edited portion of the code base is touched by the tests. *Crisp* cannot be used as a debugging tool until a stable regression test suite is available for the entire program.

Since *Crisp* applies changes at the method level, it has assumed that the sizes of methods are in general small enough such that identifying a method is helpful in the debugging process. For programs where methods are monolithic and changes occur only in a few places, *Crisp* may provide little assistance by identifying a changed method as failure-inducing.

Since *Chianti* only decomposes edits of the source code, both *Crisp* and *Chianti* can identify affecting changes and failure-inducing changes based on the changes in the source code. Changes to the test inputs, the configuration of the program, as well as the execution environment are not captured under this methodology. In fact, some of the failing tests discovered are excluded from our experimental results because the failures were not caused by the source code edit.

7.2 Failure-Inducing Changes Versus Bugs

Being able to pinpoint where unexpected behavior manifests itself does not necessarily mean that the program bugs are located in the same place. Many approaches have been proposed, which are discussed in the next chapter, to locate software bugs. Most of these approaches do not scale to large programs and/or report significant false positives. *Crisp* provides a mechanism for focusing programmers on a few places where they can begin a detailed analysis. However the debugging process should not stop after *Crisp* suggests the set of failure-inducing changes. On the other hand, unexpected behavior does not necessarily translate into undesirable behavior. During our testing of *Crisp*, only a few versions of `Daikon` and `Eclipse jdt.core` contained failing tests from the repository; the reason being that some of the tests were altered from one version to the other. This indicates that in practice, the regression test suite evolves with the software program. Thus, certain unexpected behavior may actually be desirable after all.

7.3 Intermediate Versions

To create compilable intermediate versions based on dependences is not trivial. In [12], we have listed a couple of limitations of *Crisp* when handling changes to the type hierarchies as well as the positioning of fields that depend on each other. Since then, we have implemented a new atomic change, ChangeTypeHierarchy (**CTD**) that captures any textual changes when a subclass extends a different superclass or implements a different interface. One of the failing tests in our experiment contains such changes and *Crisp* was successful in generating valid intermediate versions using this ChangeType-Hierarchy change and its dependences.

Dependences among changes are captured during the traversal of the abstract syntax trees of the original and edited programs; they are syntactic and fit well into the four categories of dependences defined in Section 2.2. However, the current dependences calculated by *Chianti* and augmented by *Crisp* do not capture certain semantic dependences.

The code presented in Figure 7.1 illustrates a change in the initialized values of

```
public class A {
  private int x = 1;
  ~~private int one = 1;~~
  ~~private int two = 2;~~
  private int one = 2;
  private int two = 1;
  public String foo() {
    switch(x) {
      case one:  return ‘‘one’’;
      case two:  return ‘‘two’’;
      default:  ‘‘none’’; }
  }
}
public class TestA extends TestCase {
  public void test() {
    A a = new A();
    Assert.assertEquals(a.foo(),’’one’’); }
}
```

Figure 7.1: Original Program (contains the struck out code) and Edited Program (contains the boxed code)

member fields `A.one` and `A.two`. `TestA.test()` passes in the original program but fails in the edited one. Although there is no code change within the method `A.foo()`, *Chianti* accurately captures the affecting changes **CFI**(A.one) and **CFI**(A.two) by using the mapping dependences between `A.one`, `A.two` and A's implicit constructor `init`(). However, none of the dependence categories captures the value dependences between `A.one` and `A.two`. `A.one` cannot change its value from 1 to 2 without the compiler complaining that there are duplicate cases among the case statements. These two changes need to be applied to the original program together, but understanding their semantic relationships is beyond our current methodology.

Chapter 8

Related Work

This chapter summarizes related work in fault localization that had been extensively discussed in [12]. For related work in change impact analysis and *Chianti*, please refer to [15], [14], [13], [5] as well as [12].

Other areas of research relevant for comparison with *Crisp* are delta debugging, techniques for avoiding recompilation, and fault localization.

8.1 Delta Debugging

In the work by Zeller et al. on *delta debugging*, the reason for a program failure is identified as a set of differences between program versions [18] that distinguish a succeeding program execution from a failing one.

A set of failure-inducing differences is determined by repeatedly applying different subsets of the changes to the original program and observing the outcome of executing the resulting intermediate programs. By correlating the outcome of each execution, with the set of changes applied, one can narrow down the set of changes responsible for the failure. Delta debugging has been applied successfully to very large programs [18].

In the examination of differences between program versions, both delta debugging and our work aim at identifying failure-inducing changes; however, there are several important differences between the two approaches. First, delta debugging searches the entire set of changes to find the failure-inducing changes. In our approach, we first obtain the set of affecting changes for a failed test with *Chianti*, and then generate the intermediate versions of programs just from this small set of changes. By associating each test with its corresponding affecting changes, a large set of uncorrelated changes can be ignored, so that a programmer can focus on only those changes related to the given test. This is extremely useful when re-execution of the regression test suites is costly. Second, delta debugging builds the intermediate versions by only using the structural differences between succeeding and failing program executions (e.g., changing one line or one character to generate an intermediate program version) and it is language-independent. Our model of dependences between atomic changes ensures that *Crisp* only builds meaningful intermediate versions of Java programs, which reduces the number of intermediate programs that need to be constructed. When a programmer selects a set of interesting changes, *Crisp* automatically augments these changes with all the prerequisites necessary to build a syntactically valid program version. Locating multiple failure-inducing changes using delta debugging has exponential complexity in the worst case, since all combinations of changes need to be explored one by one. By using change dependences and complement sets, *Crisp* is able to locate multiple failure-inducing changes efficiently. Unlike delta debugging, *Crisp* cannot guarantee the identification of a minimal set of failure-inducing changes due to the presence of inter-dependent failure-inducing changes.

8.2 Techniques for Avoiding Recompilation

Existing techniques to avoid unnecessary recompilation use dependences between compilation units of a program to calculate which other units (i.e., clients) might require recompilation. This may be necessary, for example, if a specific compilation unit that defines functions or types is changed. This calculation uses inter-unit dependences that can be supplied by the programmer (i.e., as in the UNIX *make* [8]) or based on derived syntactic or semantic relationships. These dependences, describing clients of changed program constructs, are *incomparable* to the dependences used in *Crisp* that capture necessary additions to user-selected fine-grained changes required to form a minimal syntactically valid edit, because each captures different information.

Here, we summarize briefly several approaches to avoiding recompilation as representative of this research area. These techniques differ in their definitions of dependence and the granularity of the compilation units used, (i.e., files, classes or modules [4, 10]).

The earliest work was *smart recompilation* by Tichy [16, 1] which defined dependences between compilation units, induced by *Pascal include* files that contained global constants and type definitions. Syntactic dependences were constructed between *include* files and those Pascal code files (i.e., $*.p$ files) which contained references to the include-defined constructs (e.g., types, constants). Tichy et. al later compared several smart recompilation approaches [1] empirically to quantify their benefits on several Ada programs, finding a savings of approximately 50% of the recompilation effort. Burke and Torczon [3] described semantic dependences between procedures derived from interprocedural dataflow information for Fortran programs. Their dependences were calculated using the alias, side-effect, reference and constant-value information associated with each subroutine, assuming that this information might have been used to enable optimizations during compilation. Their technique was capable of fine-grained recompilation decisions on a procedure level. More recently, Dmitriev [7] used information provided in Java class files to calculate syntactic dependences between program constructs (e.g., fields, methods). His approach, called *smart dependency checking*, was to aggregate these dependences in order to ascertain the clients of a class (i.e., classes referencing members of another class). Thus, when the code for a class changes, its client classes are marked for recompilation. This automates the creation of dependences which can be used with *make* for Java programs.

8.3 Fault Localization

Program slicing [17] has been suggested as a technique for localizing faults: Computing a slice with respect to an incorrect value determines all statements that may have contributed to that value, and will generally include the statement(s) that contain the error. Since slices may become very large, techniques such as *program dicing* [11] have been developed, where a slice with respect to an erroneous value is intersected with a slice with respect to a correct value. DeMillo et. al. [6] suggest *critical slicing* as a technique to localize faults in a program. A statement is critical if, without it program execution reaches the same failure statement s_F, but with different values for referenced variables. They report that their technique is able to reduce relevant program size by around 64% and retain the failure-inducing statement in 80% of the cases. Bunus and Fritzson [2] suggest a semi-automatic debugging framework for equation-based languages used to model physical systems. Their approach uses program slicing and dicing on a combination of execution traces, dependence graphs and assertions to help programmers find and correct bugs in an interactive debugging session. Gupta et. al. [9] uses delta debugging to *simplify* or *isolate* inputs that are failure-inducing, and then uses forward and backward dynamic slices to suggest a set of statements that could potentially contain the fault.

There are two major differences between our approach and slicing's approach to finding faults. Program slicing is a fine-grained analysis at the statement level that can be used to inspect a failing program to help locate the cause of the failure. Our work focuses on failures that occur due to a specific edit between program versions, and our analysis is at the method level.

Chapter 9

Conclusion

This thesis describes *Crisp*, a tool to automatically construct intermediate versions of a program that has been edited, in order to find those parts of the edit that have caused a test method within a test suite to fail. *Crisp* is built using results generated from a change impact analysis tool, *Chianti*. *Chianti* provides *Crisp* with a set of *affecting changes* that affects the behavior of a failing test as well as the dependences among these changes. Then, *Crisp* automatically constructs the *to-be-applied* lists for these affecting changes. In the semi-automatic mode, the programmer is provided with a graphical user interface where she can choose interesting affecting changes to be applied to the original program, thereby creating various intermediate versions of the program. Each of these intermediate versions is free of compilation error and can be exercised by the failing test. *Crisp* also automatically calculates the complement set for a given set of failure-inducing changes. The complement set can be applied to the original program to determine whether all the failure-inducing changes have been identified. Finally, the implementation of an exploration algorithm in *Crisp* provides an automatic mode for programmers in which the set of failure-inducing changes can be determined with minimal user input. This eliminates the tedious task of exploring individual change when the affecting changes set is large and there may be more than one failure-inducing changes.

Experiments using *Crisp* have been performed on two moderate-sized Java programs, `Daikon` and the `Eclipse jdt.core`. Among the versions extracted from the repository history of these two programs, 31 unique failing tests were identified. Using *Crisp*, failure-inducing changes were identified for these failing tests without any prior knowledge of the source code. The size of the failure-inducing changes set ranged from one to eight. Twenty-eight out of 31 failing tests had one or two failure-inducing changes. On average, the size of the failure-inducing changes set is about 5.7% of the size of the editable changes set provided by *Crisp*, which in turn has been reduced by 25% from the original affecting changes set provided by *Chianti*. Thus, *Crisp* can achieve an order of magnitude reduction in the number of changes to be examined, a critical result to the programmer who is dealing with large code base. From a performance standpoint, the automatic algorithm has provided response time that is consistent with the execution time of the failing test; however, the performance of the algorithm can be further improved by an effective ordering heuristic. The results of the experiments also reveal some of the limitations of syntactic dependences. Investigation into the semantic dependences among the affecting changes will be part of the future work.

References

[1] R. Adams, W. F. Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering Methodology*, 3(1):3–28, 1994.

[2] P. Bunus and P. Fritzson. Semi-automatic fault localization and behavior verification for physical system simulation models. In *In Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 253–258, Montreal, Quebec, Canada, October 2003.

[3] M. Burke and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, 1993.

[4] C. Chambers, J. Dean, and D. Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 221–230, 1995.

[5] Ophelia Chesley, Xiaoxia Ren, and Barbara G. Ryder. Crisp: A debugging tool for Java programs. In *Proc. of the Int. Conf. on Software Maintentance*, pages 401–410, September 2005.

[6] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 121–134, New York, NY, USA, 1996. ACM Press.

[7] M. Dmitriev. Language-specific make technology for the java programming language. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 373–385, 2002.

[8] Stuart I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–65, 1979.

[9] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proc. of the Int. Conf. on Automated Software Engineering*, pages 263 – 272, 2005.

[10] R. Hood, K. Kennedy, and H. A. Muller. Efficient recompilation of module interfaces in a software development environment. In *SDE 2: Proceedings of the second ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 180–189, 1987.

[11] J.R. Lyle and M. Weiser. Automatic bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, pages 877–883, Beijing (Peking), China, 1987.

[12] Xiaoxia Ren, Ophelia C. Chesley, and Barbara G. Ryder. Identifying failure causes in java programs: An application of change impact analysis. *IEEE Transactions on Software Engineering*, 32(9):718–732, 2006.

[13] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proc. of the ACM SIGPLAN Conf. on Object Oriented Programming Languages and Systems (OOPSLA'04)*, pages 432–448, Vancouver, Canada, October 2004.

[14] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, Ophelia Chesley, and Julian Dolby. Chianti: A prototype change impact analysis tool for Java. Technical Report DCS-TR-533, Rutgers University Department of Computer Science, September 2003.

[15] Barbara G. Ryder and Frank Tip. Change impact for object oriented programs. In *Proc. of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis and Software Testing (PASTE01)*, June 2001.

[16] W. F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, 1986.

[17] Frank Tip. A survey of program slicing techniques. *J. of Programming Languages*, 3(3):121–189, 1995.

[18] Andreas Zeller. 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)*, pages 253–267, Toulouse, France, 1999.