Enhancing Traditional Behavioral Testing through Program Slicing

Philippe Charland

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

September 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-94736-X
Our file Notre référence
ISBN: 0-612-94736-X

ABSTRACT

Enhancing Traditional Behavioral Testing through Program Slicing

Philippe Charland

Although there has been much research on the application of program slicing to the problem of software testing, most of it has focussed on regression testing. The objective of the published techniques is to reduce its cost by identifying the set of existing test cases which are guaranteed to exercise the modified program components. In this research, program slicing is applied to behavioral testing. Three testing approaches are presented to ensure that modifications made to a program have not adversely affected its correct behavior. The proposed testing techniques, as well as the underlying dynamic program slicing algorithm, are implemented as part of the CONCEPT research project. A case study using the JUnit testing framework is also presented to demonstrate their applicability in detecting faults, which could escape from traditional testing techniques.

ACKNOWLEDGEMENTS

I would first like to thank my supervisor, Dr. Juergen Rilling, for his support, guidance, and patience over the past two years. He was a constant source of helpful ideas, constructive comments, and judicious advice.

I would also like to thank all the members of the CONCEPT research group, with whom I had useful discussions while working on this research. I am particularly indebted to my colleague Yonggang Zhang, for his work on the parser.

Finally, I would like to thank my parents and family for their support and encouragements. In particular, I would like to express my gratitude to Michel Charland, for having patiently proofread this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Software development consists of a series of production activities wherein numerous opportunities for introducing faults exist. Errors may begin to occur at the very inception of the process, when the objectives are erroneously or imperfectly specified, or in later design and development stages [DEU82].

Since humans are generally incapable of developing software without making errors, the generated source code has to be tested to uncover as many of those as possible, before the product is delivered to the client. A rich variety of testing techniques have evolved for software. These "provide systematic guidance for designing tests that (1) exercise the internal logic of software components, and (2) exercise the input and output domains of the program to uncover errors in program function, behavior, and performance" [PRE01].

Every time a major change is made to a software system, including the integration of a new component, some of the tests which have already been conducted have to be re-executed. This is to ensure that the modifications have not propagated unintended side effects. Program slicing, a program reduction technique, has been used in several published techniques to identify the set of existing tests which will exercise the modified functionalities [AGR93B, BAT93, BINK97, GUP92, and ROT94]. Once this set is identified, the tests are run on the modified version of the software system and the actual results are compared with the expected ones.

Relying solely on the comparison between the actual and required results may not be sufficient to ensure that no unintended behavior or additional faults were introduced. Faulty code can occasionally produce correct behavior. For example, if $x + x$ is incorrectly coded instead of $x * x$, when x is 2, the correct result is produced. Although only a little more testing is required to reveal the fault in this case, simple errors can sometimes result in very pernicious fault hiding [BIN00].

In this research, program slicing is used to facilitate behavioral testing. Three testing techniques are proposed to ensure that the modifications made to a software system have not changed the correct behavior inherited from the original version. These approaches are also implemented as part of the CONCEPT research project.

The remainder of this thesis is organized as follows: Section 2 introduces some relevant software testing and program slicing concepts. It also reviews research on the application of program slicing to the problem of software testing. In Section 3, the different behavioral testing techniques are presented. Their implementation is discussed in Section 4. Section 5 describes a case study performed using JUnit [BEC98] to demonstrate their applicability. Finally, Section 6 provides the conclusions and future work.

2. BACKGROUND

2.1 SOFTWARE TESTING

Software testing is the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of it [IEE87A]. It is a critical activity of the software process, as it "represents the ultimate review of specification, design, and code generation" [PRE01]. Software testing consumes at least half of the labor expended to produce a working program [BEI90, KUN98]. Different studies indicate that between 1 and 3 errors are made per 100 lines of source code [BEI90, VAN00]. The objectives of software testing are to prevent and discover as many of those faults as possible, at the lowest cost.

To achieve the above objectives, test cases are designed. A test case is "a set of inputs, execution conditions, and expected results developed for a particular objective" [BIN00]. A collection of test cases related by a testing goal or an implementation dependency is called a test suite [BIN00].

A successful test case is one which reveals a fault, rather than shows that the program under test works as expected. Test cases can be designed from a functional or structural perspective.

2.1.1 Functional Testing

In functional testing, also called black-box or behavioral testing, "test cases are derived from the specification of the software" [VAN00]. As illustrated in Figure 2.1.1.1, the system is treated as a black box, whose behavior can only be determined by studying its inputs and the related outputs [SOM01]. Following are some functional testing methods.

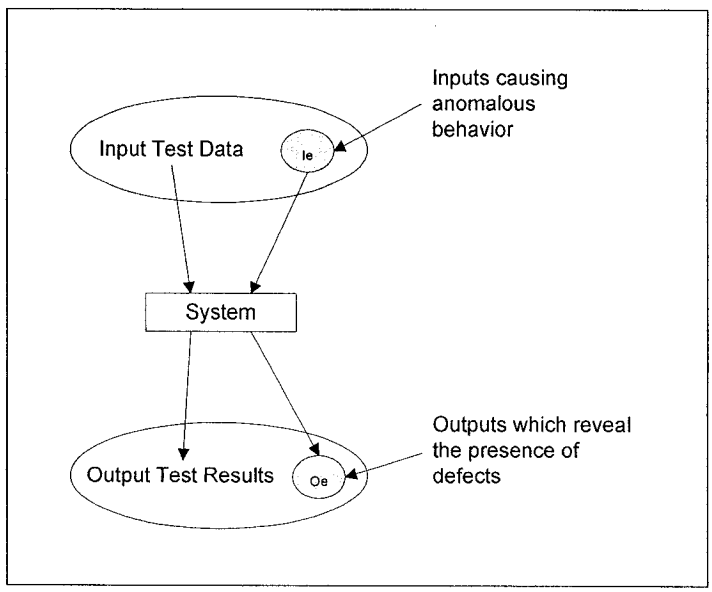

Figure 2.1.1.1 Black-Box Testing

INPUT DOMAIN TESTING. Consists of extremal and midrange testing. In extremal testing, "test data are chosen to cover the extremes of the input domain" [MOR92]. For midrange testing, test data are selected from the interior of the domain [MOR92]. The motivation of input domain testing is inductive: it is hoped that conclusions about the entire input domain can be drawn from the behavior elicited by some of its representative members [MYE79].

EQUIVALENCE PARTITIONING. Usually, the input domain of a program can be partitioned into a number of different classes, which have common characteristics. Equivalence partitioning "divides the input domain of a program into classes of data from which test cases can be derived" [PRE01]. Extremal and midrange testing are then applied on the resulting input subdomains. A fault which affects the behavior within a subdomain is a computation fault, whereas one which affects the boundaries of a subdomain is a domain fault [HIE02].

In Figure 2.1.1.2, each equivalence class is shown as an ellipse [SOM01]. Equivalence partitioning attempts to define test cases which reveal classes of faults, thereby reducing the number of test cases which need to be designed.

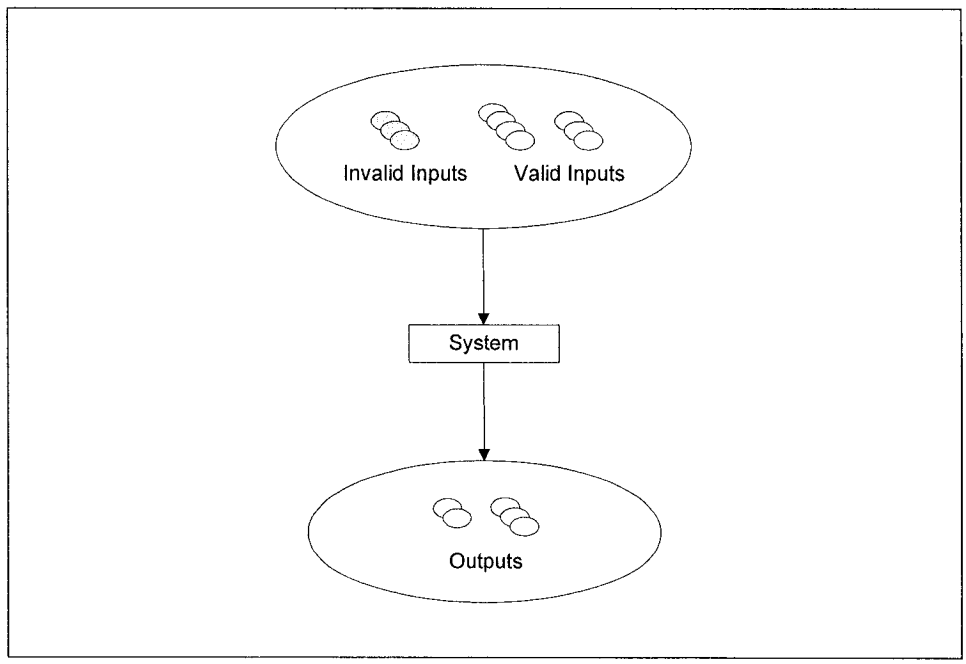

Figure 2.1.1.2 Equivalence Partitioning

In the case of a program which accepts 4 to 10 inputs which are five-digit integers greater than 10,000, the equivalence partitions and possible test input values are illustrated in Figure 2.1.1.3 [SOM01].

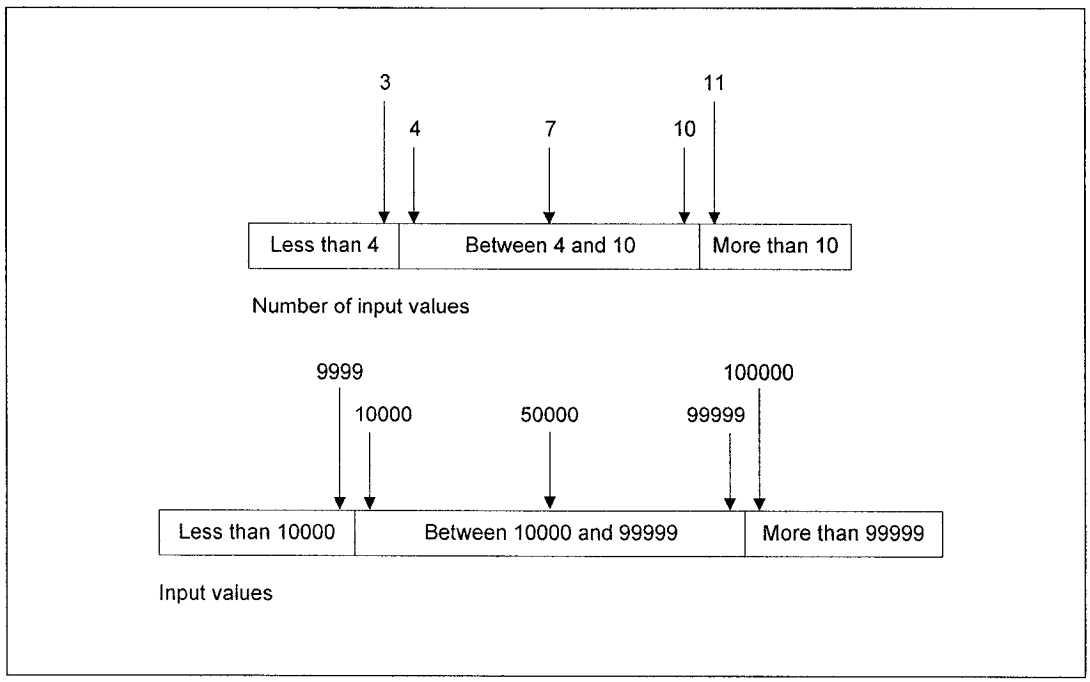

Figure 2.1.1.3 Equivalence Classes

SYNTAX CHECKING. Verifies that the program "parses its input and handles incorrectly formatted data" [BEI90]. This is accomplished by executing the program using a broad spectrum of test cases, some of which violate the input format rules.

SPECIAL VALUE TESTING. Selects test cases "on the basis of features of the function to be computed" [HOW80]. Special value testing is mostly used in the case of mathematical computations. An example would be to test a factorial function with input value 0.

OUTPUT DOMAIN COVERAGE. As illustrated in Figure 2.1.1.2, "for each function determined by equivalence partitioning, there is an associated output domain" [MOR92]. Output domain coverage is performed by selecting test data that will cause the extremes of each of the output domains to be achieved [HOW80]. This guarantees that the minimum and maximum output conditions have been verified and all categories of error messages have been generated.

2.1.2 Structural Testing

Structural testing, sometimes called white-box or glass-box testing, "implies inspection of the source code of the program and selection of test cases that together cover the program, as opposed to its specifications" [PER90]. Next are some structural coverage measures.

STATEMENT TESTING. Statement coverage is achieved when every statement in the program has been executed at least once [BIN00]. It is the minimum required by the IEEE software engineering standards [IEE87B]. If there is a fault in a statement and this statement is not executed, then it is almost impossible that this fault will be uncovered. However, statement testing is a very weak criterion and Beizer argues that "testing less than this for new software is unconscionable and should be criminalized" [BEI90].

BRANCH TESTING. Branch coverage is achieved when every path from a predicate statement is executed at least once by a test suite [BIN00]. It is an improvement over statement coverage, but considers compound predicates as single statements.

CONDITIONAL TESTING. Condition coverage subsumes branch coverage. It "requires that all true-false combinations of simple conditions be exercised at least once" [BIN00]. As a result, for a predicate statement consisting of n simple conditions, there are 2^n true-false combinations.

PATH TESTING. The objective of path testing "is to exercise every independent execution path through a component or program" [SOM01]. An independent path is any path through the program that introduces at least one new predicate statement. If every independent path is executed, then statement and branch coverage are achieved.

The number of independent paths to execute is determined by computing the cyclomatic complexity C of the program flow graph. A flow graph consists of nodes and edges, which respectively represent predicate statements and flow of control. Figure 2.1.1.5 shows the flow graph of the binary search procedure displayed in Figure 2.1.1.4 [SOM01]. C is computed according to the following formula: $C = e - n + 2$, where e is the number of edges and n, the number of nodes in the flow graph.

```
   class BinSearch {

1    public static void search(int key, int [] elemArray, Result r) {
       int bottom = 0;
       int top = elemArray.length - 1;
       int mid;
       r.found = false;
       f.index = -1;

2      while(bottom <= top) {
         mid = (top + bottom) / 2;

3        if (elemArray[mid] == key) {
           r.index = mid;
           r.found = true;
           return;
         }

         else {
4          if (elemArray[mid] < key)
5            bottom = mid + 1;
           else
6            top = mid - 1;
7        }
8      }
9    }
   }
```

Figure 2.1.1.4 Sample Program

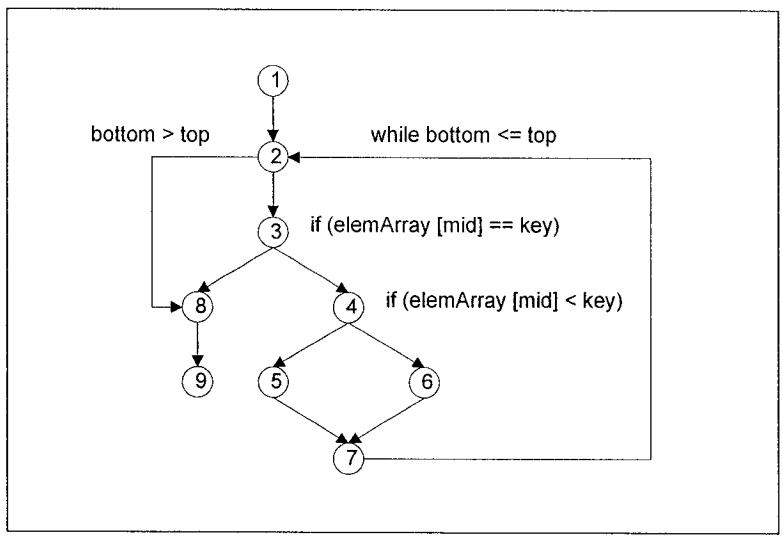

Figure 2.1.1.5 Flow Graph of the Sample Program

The cyclomatic complexity of the binary search routine is 4. Its set of independent paths which need to be executed are:

Path 1: 1, 2, 3, 8, 9
Path 2: 1, 2, 3, 4, 6, 7, 2
Path 3: 1, 2, 3, 4, 5, 7, 2
Path 4: 1, 2, 3, 4, 6, 7, 2, 8, 9

DATA FLOW TESTING. Data flow testing "selects test paths of a program according to the locations of definitions and uses of variables in a program" [PRE01]. A definition of variable *v* is a statement which assigns a value to that variable. A use of variable *v* is a statement in which this variable is referenced. Several variations of data flow coverage have been proposed and ranked [FRA88, FRA93]. One level of coverage, referred as the all-uses criterion, has been found to be effective. It requires that at least one definition-use (DU) path be exercised for every DU pair. A DU path of variable *v* is a tuple (*v*, *i*, *j*), where (1) *i* and *j* are program statements, (2) *v* is defined at *i*, (3) *v* is used at *j*, and (4) there exists a path from *i* to *j* without an intervening definition of variable *v*.

LOOP TESTING. Loop testing "focuses exclusively on the validity of loop constructs" [PRE01]. Coverage requirements depend on the kind of loop defined. As illustrated in Figure 2.1.1.6, there are four different classes of loops: simple, concatenated, nested, and unstructured [BEI90]. A minimum test suite for a loop with variable iteration control bypasses the loop (zero iteration) and exercises it with an iteration [MAR94]. Two iterations are the minimum needed to detect data initialization and use faults [FRA88]. Loop control boundary conditions, a frequent source of control faults, require more

extensive coverage. Time should not be wasted testing unstructured loops: they should be redesigned.

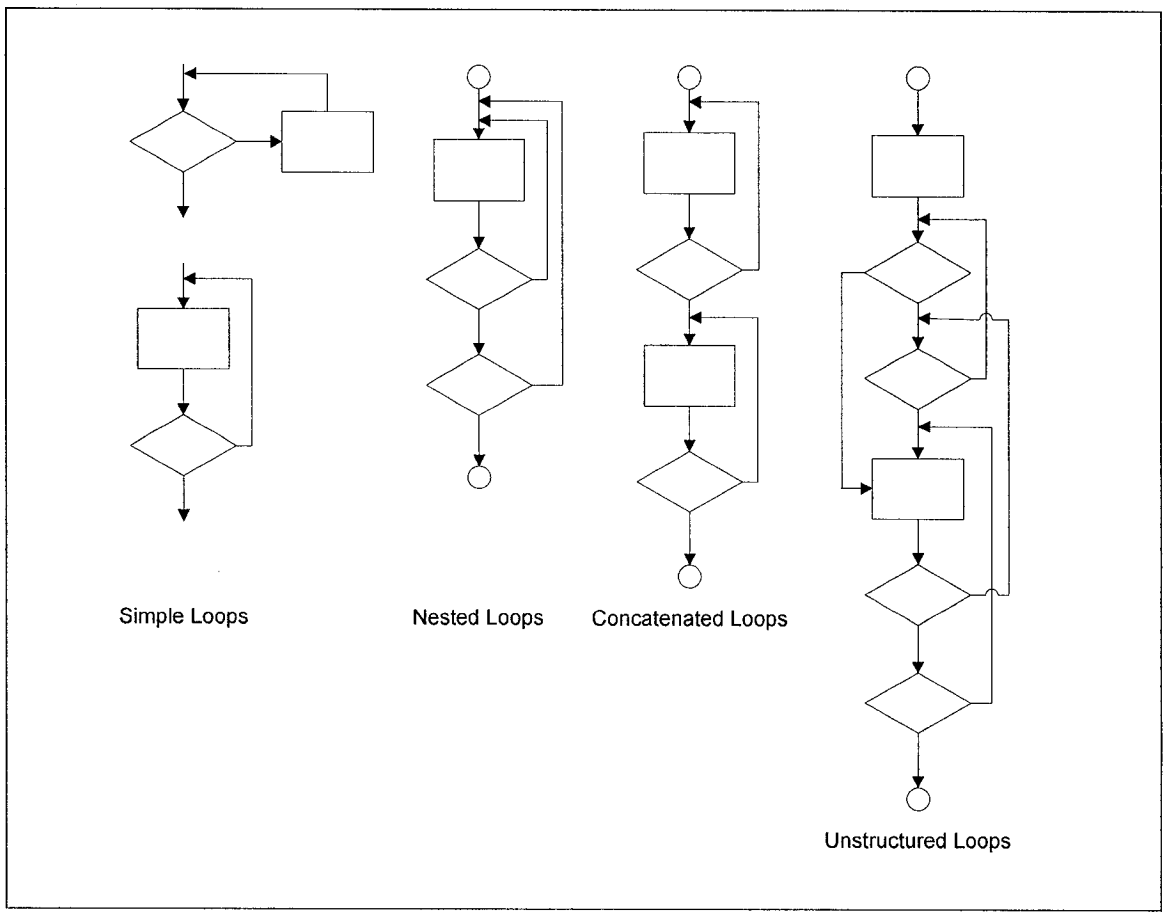

Figure 2.1.1.6 Classes of Loops

2.2 Regression Testing

Regression testing can be defined as "any repetition of tests (usually after a software or data change) intended to show that the software's behavior is unchanged except insofar as required by the change to the software or data" [BEI90]. Its goal is to provide confidence that the modified program behaves as intended and ensure that the fault fixes

and new functionalities have not changed unintentionally the correct behavior inherited from the original program [BINK98].

There are two approaches to regression testing: retest-all and selective retest. In retest-all, every test which has been previously executed is rerun. Since this is usually too expensive both in terms of time and effort, one can opt for a selective retest. This technique consists of rerunning only the test cases for which the original and modified programs might produce different results. This avoids the costly re-execution of a subset of the test suite, since it omits all the test cases which are guaranteed to generate the same outcomes.

2.3 Object-Oriented Software Testing

2.3.1 Motivation

Object-oriented software testing deals with the new problems introduced by the features of object-oriented languages [KUN98]. Encapsulation, inheritance, polymorphism, and dynamic binding provide visible benefits in software design and programming, but raise at the same time new challenging problems in the software testing and maintenance phases [BIN94, LEJ92, and WIL92].

Encapsulation is the mechanism which binds together inside an object the operations and attributes it manipulates. This makes the interaction between two or more objects

implicit in the code and as a result, complicates their understanding as well as the preparation of test cases to exercise such interactions [LET86].

Inheritance allows a class to inherit attributes and operations from a base class and therefore, promotes code reuse. However, an operation which was tested to be "correct" in the context of the base class does not guarantee that it will work "correctly" in the context of the derived class [PER90].

Polymorphism is the mechanism which allows one operation name to be associated with different operations. The version of the operation appropriate to the situation is selected at run time, a process called dynamic binding. These features make testing more difficult, because the exact implementation cannot be determined statically and therefore, the control flow of a program is less transparent [SMI90].

Despite the problems outlined above, nearly everything learned about testing procedural language programs also applies to object-oriented testing. It is the emphasis and effectiveness of the various test techniques which differ for object-oriented programming [BIN00]. One example which illustrates this is the case of object state testing.

2.3.2 Object State Testing

Object state testing focuses on testing the state dependent behaviors of objects [KUN96]. In procedural programming, state dependent behaviors are normally found in embedded

systems. On the other hand, in object-oriented programming, a large number of objects have state dependent behaviors and these objects can have an effect upon each other. Object state testing is therefore a crucial aspect of object-oriented testing.

State dependent behavior signifies that the result of a method call on an object can depend on its state, i.e., the combination of the values of its instance variables, or the state of other objects. It also implies that the execution of a method can cause state changes to more than one object.

The purpose of object state testing is to test the composite effects of method calls on objects so as to identify any sequences of those which can lead to invalid states. Object state testing designs test cases "by modeling the system under test as a state machine" [BIN00]. A state machine is a system model composed of events, states, and actions. An action is determined by the current and past events and the effects of previous events are represented by states [BIN00].

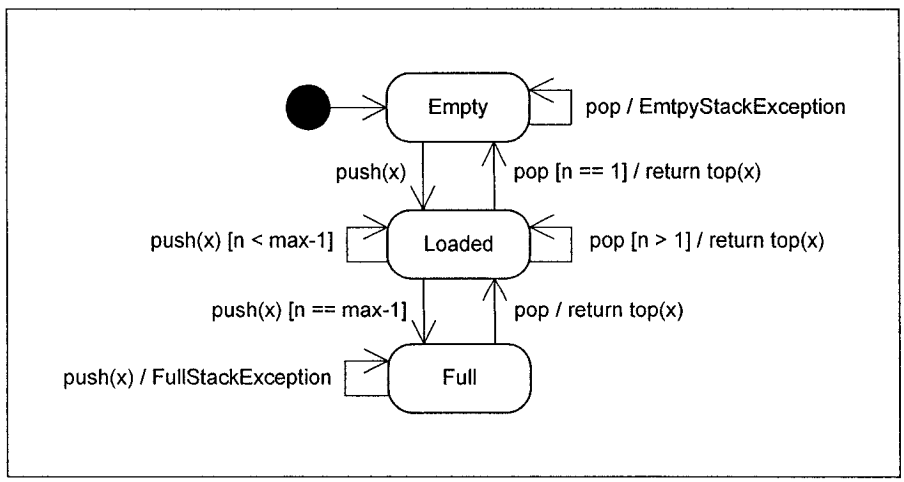

Figure 2.3.2.1 State Machine of a Stack

Figure 2.3.2.1 models the state machine of a stack [BIN00]. Test suites which are very effective at finding faults can be generated from state machines. For example, adequate object state testing would have revealed the fault in the following integer set class, which was part of a commercial C++ library [HOF95].

```
class IntSet {
  public:
    // operations on single sets
    IntSet();
    ~IntSet();
    IntSet& add(int);          // Add a member
    IntSet& remove(int);       // Remove a member
    IntSet& clear();           // Remove all members
    int     is_member(int);    // Is arg a member?
    int     extent();          // Number of elements
    int     is_empty();        // Empty or not?

    // operations on pairs of sets ...
};
```

Figure 2.3.2.2 Integer Set Class

In the above class, the `add(n)` function throws a Duplicate exception if `n` is already present in the set. To test this exception, `add(1)` was invoked twice. Even though the exception was thrown correctly on the second call, a duplicate value of the element was still added to the set. The fault escaped detection when a subsequent `remove(1)` was invoked without generating an exception. The fault was later uncovered when it was noticed that two invocations of `remove(n)` were necessary for `is_member()` to return false, after having called `add(n)` twice.

2.4 PROGRAM SLICING

Program slicing is a program reduction technique which allows one to narrow down the size of the source code of interest by identifying only the parts of a program that are relevant to the computation of a particular variable [RIL01A]. Program slicing techniques can be classified as static or dynamic.

2.4.1 Static Program Slicing

The notion of static program slicing originated in the seminal paper by Weiser [WEI82, WEI84], who defined a slice S as a reduced, executable program obtained from a program P by removing statements such that S replicates parts of the behavior of P. In a more formal way, a static program slice consists of the parts of a program P that could potentially affect the value of a variable v at a point of interest p. The static program slicing algorithm of Weiser computes slices using information derived from the source code. Different extensions of the original static slicing approach have been proposed, e.g., [CHU02, HAR97, HAR01A, HAR01B, KRI94, LAR96, and LAW94].

A static slice is defined for a slicing criterion of the form $C = (x, V)$, where x is a statement in program P and V is a subset of the variables in P. Given C, the program slice consists of all the statements in P which could potentially affect variables in V at position x for the set of all possible inputs. Static slices are computed by finding

consecutive sets of indirectly relevant statements, according to the data and control dependencies of a program dependence graph.

2.4.2 Program Dependence Graph

The concept of program dependence graph (PDG) was originally defined by Ottenstein and Ottenstein [OTT84] and later refined by Horwitz et al. [HOR90, REP88, and REP89]. It is a directed graph $G = (N, A, s, e)$, where (1) N is a set of nodes, (2) A, a set of arcs, is a binary relation on N, and (3) s and e are respectively the unique entry and exit nodes [KOR97A]. A path from the entry node s to some node k, $k \in N$, is a sequence $<n_1, n_2, \ldots, n_q>$ of nodes such that $n_1 = s$, $n_q = k$ and $(n_i, n_{i+1}) \in A$, for all n_i, $1 \leq i < q$ [KOR97A]. A path which has been executed for some input is referred as an execution trace.

A PDG is used to represent the control and data dependencies in a program. A node corresponds to a program statement while an edge represents a control or data dependency between two nodes.

A data dependence captures the situation in which one node assigns a value to an item of data and another node uses that value [KOR97A]. It is based on the concepts of variable definition and use. Therefore, a node j is data dependent on node i if there exists a variable v such that (1) v is defined in node i, (2) v is used in node j, and (3) there exists a path from i to j without an intervening definition of variable v [RIL98].

A control dependence is based on the concept of postdominance. Informally, this can be thought of as one program statement determining in some way whether or not another statement will be executed. Ferrante [FER87] defines control dependence as follows: Let *Y* and *Z* be two nodes and *(Y, X)* be a branch of *Y*. Node *Z* postdominates node *Y* if and only if *Z* is on every path from *Y* to the exit node *e*. Node *Z* postdominates branch *(Y, X)* if and only if *Z* is on every path from *Y* to the program exit node *e* through branch *(Y, X)*. *Z* is control dependent on *Y* if and only if *Z* postdominates one of the branches of *Y* and *Z* and does not postdominate *Y*. The concept of post-dominance means that all execution paths in a control flow graph from a specific node *i* to the program end, must pass through another node *j* before they reach the program end point [HOR90, HOR92].

Figure 2.4.2.2 represents the PDG of the sample program displayed in Figure 2.4.2.1.

```
 1 class Example {
 2
 3   public static void main(String args[]) {
 4     int array[] = {1};
 5     int i   = 1;
 6     int sum = array[0];
 7
 8     while(i < array.length) {
 9       sum += array[i];
10       i++;
11     }
12
13     System.out.println(i);
14     System.out.println(sum);
15   }
16 }
```

Figure 2.4.2.1 Sample Program

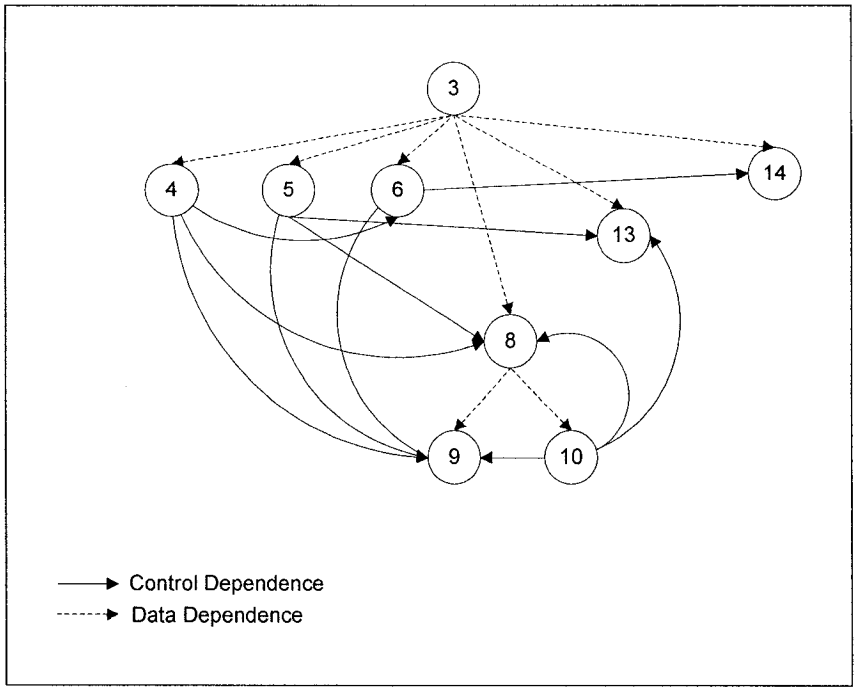

Figure 2.4.2.2 PDG of the Sample Program

PDGs can be used for the computation of static slices. The static slice S of a program P, with respect to variable v at position i, is obtained by traversing the PDG of P backwards along its edges, starting from node i. The nodes, which were visited during the traversal, constitute S [OTT84]. Figure 2.4.2.3 shows the static slice for variable *sum* of the sample program at position 14.

```
 1 class Example {
 2
 3   public static void main(String args[]) {
 4     int array[] = {1};
 5     int i   = 1;
 6     int sum = array[0];
 7
 8     while(i < array.length) {
 9       sum += array[i];
10       i++;
11     }
12
14     System.out.println(sum);
15   }
16 }
```

Figure 2.4.2.3 Static Slice for Variable *sum* at Statement 14

2.4.3 Dynamic Program Slicing

Korel and Laski introduced in [KOR88] the notion of dynamic slicing. For this approach, not only static information is used, but also dynamic information regarding the program execution for some specific program input. A dynamic slice preserves the program behavior for a specific input, in contrast to a static slice, which preserves the program behavior for the set of all inputs for which the program terminates. As a result, by considering only a particular program execution instead of all possible ones, dynamic algorithms tend to compute significantly smaller slices than static algorithms.

A dynamic slicing criterion is a tuple $C = (x, y^q)$, where x is the program input on which a program P was executed and y^q is a variable y at execution position q. A dynamic slice of a program P for the slicing criterion C is any syntactically correct and executable program P', which is obtained from P by deleting zero or more statements. Furthermore, when executed on program input x, it produces an execution trace T'_x, for which there exists the corresponding execution position q', such that the value of y^q in T_x equals the value of $y^{q'}$ in T_x' [KOR97A]. Figure 2.4.3.1 shows the dynamic slice for variable *sum* of the sample program at position 14.

```
 1 class Example {
 2
 3   public static void main(String args[]) {
 4     int array[] = {1};
 6     int sum = array[0];
 7
14     System.out.println(sum);
15   }
16 }
```

Figure 2.4.3.1 Dynamic Slice for Variable *sum* at Statement 14

To perform dynamic slicing, two major approaches have evolved. Backward algorithms [AGR94, GOP91, KAM93B, and ZHA98] trace backwards a recorded execution trace to derive data and control dependencies that are then used for the computation of the dynamic slice. In contrast, forward algorithms [KOR94] aim to overcome a major weakness of the backward approach: the necessity of recording the execution trace during program execution.

2.4.4 Applications of Program Slicing

When the original concept of program slicing was introduced by Weiser, its principal application was debugging [WEI82, WEI84]. If a program computes an incorrect value for a variable v at position p, the fault is expected to be found in the slice with respect to v at that point. The use of program slicing for debugging has been further explored in [AGR93A, CHO91, FRI92, LYL87, and PAN93].

Other applications of program slicing have since been proposed. For example, an inherent problem in software maintenance consists of determining whether or not a change at one point in the program will affect the behavior of other parts. In [GAL91], Gallagher and Lyle use static slicing to decompose a program into a set of components, each of which captures part of the original program's behavior. They present a collection of rules for the maintainer of a component that if respected, ensures that the changes are completely contained inside the component. Furthermore, they describe how the changes can be merged back into the complete program in a semantically consistent way.

Larus and Chandra have applied program slicing to compiler tuning [LAR94]. In their approach, dynamic slicing is used to detect potential occurrences of redundant common subexpressions, which indicate the generation of sub-optimal code.

Program slicing has also been used to parallelize the execution of sequential programs [WEI83], in program comprehension [KOR97B, RIL01B], reverse engineering [BEC93, JAC94A, and JAC94B] and testing, the latter being covered in more detail in the next section.

2.5 Program Slicing and Software Testing

Since program slicing is a source code reduction technique, it has mostly been applied for structural testing. However, it can also be used in functional testing. For example, given a functional test case, "an executable slice that captures the functionality to be tested should be easier and less expensive to test than the complete program [BINK98]."

2.5.1 Data Flow Testing

In data flow testing, a program satisfies a conventional coverage criterion if every DU pair is exercised at least once by a successful test case. In [DUE92], Duesterwald et al. propose a more rigorous testing criterion based on program slicing. Besides having to satisfy the previous criterion, each DU pair must also be output-influencing. A DU pair is output-influencing, i.e., has an influence on at least one output value, if it occurs in an

output slice, a slice with respect to an output statement. The proposed data flow testing criterion uses three slicing approaches: static, dynamic, and hybrid, which is a combination of the previous two.

In [KAM93A], Kamkar et al. extend the work of Duesterwald, Gupta, and Soffa to multi-procedure programs by defining the notions of interprocedural DU pairs. The interprocedural dynamic slicing method of Kamkar et al. [KAM92, KAM93B] is used to determine, for a given test case, which interprocedural DU pairs have an effect on a correct output value.

2.5.2 Regression Testing

"Testing during the maintenance phase of a program's life is dominated by regression testing" [BINK98]. This often involves running a large number of test cases on a program of a considerable size. To reduce the resulting costs, in terms of both human and machine time, several approaches utilizing program slicing have been proposed.

In [GUP92], Gupta et al. describe a data flow based regression testing method. This technique does not use slicing operators directly. It is rather based on the program slicing algorithms introduced by Weiser to explicitly detect the DU pairs that need to be retested after a modification. Only the test cases which execute the DU pairs affected by the change need to be executed again. An important benefit of this technique is that unlike

previous ones, the DU pairs that must be retested are identified without requiring the data flow history nor the recomputation of data flow for the entire program.

Agrawal et al. present in [AGR93B] an algorithm which uses relevant slicing to select the minimal subset of a test suite that must be rerun in order to test a modified program. The definition of a relevant slice makes use of the notion of potentially dependent.

The use of a variable v at location l in a given execution history is potentially dependent on an earlier occurrence p of a predicate in the execution history if (1) v is never defined between p and l but there exists another path from p to l along which v is defined, and (2) changing the evaluation of p may cause this untraversed path to be traversed [AGR93B].

A relevant slice extends a dynamic slice to include predicates on which statements in the slice are potentially dependent as well as the data and potential dependences of these predicates [BINK98].

In the algorithm of Agrawal et al. [AGR93B], a relevant slice with respect to the program's output is computed for each test case in the test suite. The modified program is then run on the test cases whose relevant slice contains a modified statement. This algorithm does not consider, however, the generation of additional test cases to cover the new functionalities of the modified program.

Bates and Horwitz introduce in [BAT93] three PDG based test-data adequacy criteria: all-vertices, all-control-edges, and all-flow-edges. Given a previously tested and modified versions of a program, their technique identifies for each criterion a safe approximation of (1) the set of statements affected by a modification, and (2) the test cases of a test suite that are guaranteed to exercise these affected statements. They were the first to make such guarantees. The set of affected components consists of the statements which were added to the previously tested version as well as any statement which has different slices for the two versions of the program. To identify the test cases which can be reused for the modified program, the statements of the two versions of the program have to be partitioned into equivalence classes; statements are in the same class if they have the same control slice [TIP95].

A control slice with respect to vertex v in PDG G is the traditional backward slice of PDG G' taken with respect to v' where (1) G' contains all the vertices of G plus a new unlabeled vertex v', and (2) G' contains all the edges of G, plus new control edges from the control-predecessors of v to v' [BINK98].

Bates and Horwitz prove that statements in the same class are exercised by the same test cases [TIP95].

The work of Bates and Horwitz considers only single procedure programs. Binkley [BINK97] presents two complementary algorithms using system dependence graphs (SDGs) and program slicing to reduce the cost of regression testing of multi-procedure

programs. The first algorithm reduces the size of the program on which test cases have to be rerun. This is accomplished by determining the set of components affected by a modification and the set of components preserved. The set of affected components is then used to produce a smaller program which captures the modified behavior of the original program. Only the test cases that execute the affected components must be rerun on this smaller and more efficient program. The second algorithm reduces the number of test cases that must be rerun for the all-vertices and all-flow-edges criteria. It is an interprocedural extension of the previous work of Bates and Horwitz.

Two other techniques based on dependence graphs make limited use of program slicing. First, Rothermel and Harrold present in [ROT94] intraprocedural and interprocedural algorithms to reduce the cost of regression testing. The first one selects the test cases that may produce different results when run on the modified version of a program. This is accomplished by performing a side-by-side walk of the original and modified programs' control dependence sub-graphs looking for nodes which have different texts. The second algorithm, which makes use of forward slicing, determines the set of DU pairs affected by a change in the modified program. In contrast to the test-case selection algorithms presented by Bates and Horwitz and by Binkley, these ones are safe in the sense that they select every test case which may produce a different output for the modified program. This is due to their treatment of deleted components. If a component, which does not affect any other, is deleted from the original program, then its tests would be selected by Rothermel and Harrold's technique, but not by the one of Bates and Horwitz and of Binkley.

The last dependence graph based technique is the Testing with Analysis and Oracle Support (TAOS) system [RIC94]. It was built to automate certain activities involved in testing a program because "the typical testing process is a human intensive activity and as such, it is usually unproductive, error prone, and often inadequately done" [RIC94]. The TAOS system represents procedures internally as program dependence graphs and can compute forward, backward, static and dynamic intraprocedural slices [BINK98]. However, the main focus of this technique is on the development and initial testing of a program. Very few details are given on its use for regression testing. Future work includes plans to develop a "proactive regression testing process that uses program dependences to determine what must be retested, what test cases and suites are available for reuse, and automatically initiates regression testing triggered by modification" [RIC94].

2.5.3 Partition Testing

Conditioned slicing is a technique to compute program slices with respect to a subset of program executions [CAN98]. It extends the notion of static slicing introduced by Weiser. A conditioned slice is defined for a slicing criterion $C = (x, V, F)$, where x is a statement in program P, V is a subset of the variables in P, and F is a condition. The latter is specified using a quantifier-free first order logic formula, which maps the set of initial program states of interest to booleans. In [HIE02], Hierons et al. demonstrate how conditioned slicing can be used to assist partition testing. In particular, they explain how conditioned slicing can determine if the uniformity hypothesis holds for a subdomain,

suggest the existence of computation and domain faults, as well as detect the existence of erroneous special cases that are not contained in the program specifications.

2.5.4 Mutation Testing

In mutation testing, a program p is changed by applying to it one or more mutation operators. Both p and the set M of mutants generated are then tested. Mutation testing is based on the notion that any test set that is capable of distinguishing a program p from programs syntactically similar to p is likely to be good at detecting faults in p [HIE99].

An input value σ kills a mutant p' if it distinguishes p from p'. A mutant p' of a program p can also be indistinguishable from p. In this case, p' is an equivalent mutant.

Even though mutation testing has proved to be highly effective to find software faults, a significant drawback currently restricts its use: only non-equivalent mutants should be used. However, even for small programs, the human effort needed to check a large number of mutants for equivalence is almost prohibitive [FRA97]. Furthermore, there will always remain a set of equivalent mutants which will remain undetected by any automated system.

In [HIE99], Hierons and Harman present a technique using program slicing to reduce the effort involved in determining whether or not mutants are equivalent. In cases where a mutant is not equivalent, it can help in finding input which kills it.

2.5.5 Robustness Testing

Some of the aspects of a program's behavior, such as its robustness, are implicit. They are not denoted by a set of variables, therefore making slicing inapplicable. To overcome this problem and capture the robustness of a subject program, Harman and Danicic [HAR95] transform it into an introspective form. Assignments to pseudo variables are added so that the program can compute aspects of its own implicit behavior. Even though this makes the program longer initially, a slice which captures the program's effect upon the pseudo variables can now be computed and used to provide an approximate answer as to whether or not the program is robust.

3. ENHANCING TRADITIONAL BEHAVIORAL TESTING

3.1 REGRESSION TESTING

After a system is changed, it has to be retested in order to verify that the "modifications have not caused unintended effects and that it still complies with its specified requirements" [IEE87A]. This process is known as regression testing. It consists of rerunning the suite of test cases which have passed on the previously verified version of the system, i.e., the baseline version, and which are expected to pass when run on the modified one.

However, the fact that the result of a test case is the same for the baseline and modified versions does not guarantee that the underlying code is exempt of faults. Faulty code can sometimes produce correct results for some inputs, a situation called coincidental correctness.

To illustrate this, consider the following example adapted from [KUN96]. The Java class `CoinBox` of Figure 3.1.1 implements the baseline version of a simple vending machine's coin box. It only accepts quarters and allows vending when two quarters are inserted.

The instance variables `qtrsCollected`, `qtrsInserted`, and `vendingEnabled` keep track respectively of the total number of quarters collected, the current number of quarters inserted, and whether or not vending is enabled. The method `insertQtr()` inserts a

quarter, `returnQtrs()` returns the quarters inserted, `vend()` gives to the customer the selected item, and `printTotal()` displays the amount of money that was inserted in the coin box.

```
 1 Class CoinBox {
 2
 3   private int     qtrsCollected;
 4   private int     qtrsInserted;
 5   private boolean vendingEnabled;
 6
 7   public CoinBox() {
 8     qtrsCollected  = 0;
 9     qtrsInserted   = 0;
10     vendingEnabled = false;
11     printTotal();
12   }
13
14   public void insertQtr() {
15     qtrsInserted++;
16
17     if (qtrsInserted > 1)
18       vendingEnabled = true;
19
20     printTotal();
21   }
22
23   public void returnQtrs() {
24     qtrsInserted   = 0;
25     vendingEnabled = false;
26     printTotal();
27   }
28
29   public void vend() {
30     if (vendingEnabled) {
31       qtrsCollected += qtrsInserted;
32       qtrsInserted   = 0;
33       vendingEnabled = false;
34       printTotal();
35     }
36   }
37
38   public void printTotal() {
39     System.out.println((float)qtrsInserted * 0.25f);
40   }
41 }
```

Figure 3.1.1 CoinBox Class - Baseline Version

Figure 3.1.2 displays the modified version of the `CoinBox` class which contains an error in its implementation. During maintenance, line 25 was inadvertently removed. As a

result, a consumer can insert two quarters, instruct the coin box to return them, and then get a free item by executing the `vend()` method.

```
 1 class CoinBox {
 2
 3   private int     qtrsCollected;
 4   private int     qtrsInserted;
 5   private boolean vendingEnabled;
 6
 7   public CoinBox() {
 8     qtrsCollected  = 0;
 9     qtrsInserted   = 0;
10     vendingEnabled = false;
11     printTotal();
12   }
13
14   public void insertQtr() {
15     qtrsInserted++;
16
17     if (qtrsInserted > 1)
18       vendingEnabled = true;
19
20     printTotal();
21   }
22
23   public void returnQtrs() {
24     qtrsInserted = 0;
25     /* vendingEnabled = false; */
26     printTotal();
27   }
28
29   public void vend() {
30     if (vendingEnabled) {
31       qtrsCollected += qtrsInserted;
32       qtrsInserted   = 0;
33       vendingEnabled = false;
34       printTotal();
35     }
36   }
37
38   public void printTotal() {
39     System.out.println((float)qtrsInserted * 0.25f);
40   }
41 }
```

Figure 3.1.2 CoinBox Class - Modified Version

Despite this fault, the baseline and modified versions produce both the required results for the test driver of Figure 3.1.3.

```
43 class Driver {
44   public static void main(String args[]) {
45     CoinBox coinBox = new CoinBox();
46     coinBox.insertQtr();
47     coinBox.insertQtr();
48     coinBox.returnQtrs();
49   }
50 }
```

Figure 3.1.3 Test Driver for the CoinBox Class

The fault in the modified version of the `CoinBox` class cannot be easily detected using traditional functional and structural testing techniques for several reasons. First of all, each method seems to implement the proper functionality. Also, the fault could hide from a test suite achieving statement and branch coverage. Furthermore, path testing would come to the conclusion that each independent path was correctly executed. However, the most important reason is that the fault is caused by interactions involving more than one method through an object state.

Three different approaches are proposed in this thesis to detect such faults and test whether or not the critical aspects of a program's behavior were changed unintentionally by a modification. The first one consists of comparing and analyzing the execution trace of the baseline and modified versions of a program for a particular test case. In the second approach, a dynamic slice is computed for a variable of interest using the execution traces generated by the test case. The resulting slices for the baseline and modified versions are then compared and analyzed. The last approach, like the previous one, consists of computing a dynamic slice for the two versions of the program. However, instead of just pinpointing all the statements which might potentially affect the value of the variable of interest, it identifies the contributing actions as well as the influencing variables at each step of the program's execution.

3.2 EXECUTION BASED TESTING

The problem associated with behavioral based testing, namely coincidental correctness, motivated the selection of the first testing approach suggested in this thesis. Execution based testing attempts to guarantee that the run time behavior of the modified version of a program is not different from its baseline version. It consists of measuring the differences between their execution traces, i.e., the sequence of statements which have been executed for a particular program input.

Baseline Version	Modified Version
44[1] public static void main(String args…)	44[1] public static void main(String args…)
45[2] CoinBox coinBox = new CoinBox();	45[2] CoinBox coinBox = new CoinBox();
7[3] public CoinBox()	7[3] public CoinBox()
8[4] qtrsCollected = 0;	8[4] qtrsCollected = 0;
9[5] qtrsInserted = 0;	9[5] qtrsInserted = 0;
10[6] vendingEnabled = false;	10[6] vendingEnabled = false;
11[7] printTotal();	11[7] printTotal();
38[8] public void printTotal()	38[8] public void printTotal()
39[9] System.out.println((float)qtrsInsert…	39[9] System.out.println((float)qtrsInsert…
46[10] coinBox.insertQtr();	46[10] coinBox.insertQtr();
14[11] public void insertQtr()	14[11] public void insertQtr()
15[12] qtrsInserted++;	15[12] qtrsInserted++;
17[13] if (qtrsInserted > 1)	17[13] if (qtrsInserted > 1)
20[14] printTotal();	20[14] printTotal();
38[15] public void printTotal()	38[15] public void printTotal()
39[16] System.out.println((float)qtrsInsert…	39[16] System.out.println((float)qtrsInsert…
47[17] coinBox.insertQtr();	47[17] coinBox.insertQtr();
14[18] public void insertQtr()	14[18] public void insertQtr()
15[19] qtrsInserted++;	15[19] qtrsInserted++;
17[20] if (qtrsInserted > 1)	17[20] if (qtrsInserted > 1)
18[21] vendingEnabled = true;	18[21] vendingEnabled = true;
20[22] printTotal();	20[22] printTotal();
38[23] public void printTotal()	38[23] public void printTotal()
39[24] System.out.println((float)qtrsInsert…	39[24] System.out.println((float)qtrsInsert…
48[25] coinBox.returnQtrs();	48[25] coinBox.returnQtrs();
23[26] public void returnQtrs()	23[26] public void returnQtrs()
24[27] qtrsInserted = 0;	24[27] qtrsInserted = 0;
25[28] **vendingEnabled = false;**	
26[29] printTotal();	26[28] printTotal();
38[30] public void printTotal()	38[29] public void printTotal()
39[31] System.out.println((float)qtrsInsert…	39[30] System.out.println((float)qtrsInsert…

Figure 3.2.1 Execution Trace of the CoinBox's Baseline and Modified Versions

Figure 3.2.1 compares the execution trace of the baseline and modified versions of the `CoinBox` class and highlights their differences in bold. In the present case, this approach

would have revealed the fault in the modified version. This is because the statement responsible for setting the instance variable `vendingEnabled` to false when the `returnQtrs()` method is invoked was not executed.

3.3 Coarse-Grained Slicing Based Testing

The second approach presented to detect run time behavioral faults involves computing a dynamic slice for both versions of a program. The resulting slices are then compared.

Baseline Version

```
 1 class CoinBox {
 2
 3   private int     qtrsCollected;
 4   private int     qtrsInserted;
 5   private boolean vendingEnabled;
 6
 7   public CoinBox() {

12   }
13
23   public void returnQtrs() {
25     vendingEnabled = false;
27   }



41 }
42
43 class Driver {
44   public static void main(...) {
45     CoinBox coinBox = new CoinBox();
48     coinBox.returnQtrs();

49   }
50 }
```

Modified Version

```
 1 class CoinBox {
 2
 3   private int     qtrsCollected;
 4   private int     qtrsInserted;
 5   private boolean vendingEnabled;
 6
 7   public CoinBox() {
 9     qtrsInserted = 0;
12   }
13
14   public void insertQtr() {
15     qtrsInserted++;
16
17     if (qtrsInserted > 1)
18       vendingEnabled = true;
21   }
41 }
42
43 class Driver {
44   public static void main(...) {
45     CoinBox coinBox = new CoinBox();
46     coinBox.insertQtr();
47     coinBox.insertQtr();
49   }
50 }
```

Figure 3.3.1 Slice of the CoinBox's Baseline and Modified Versions

Figure 3.3.1 displays the dynamic slice of the baseline and modified versions of the `CoinBox` class. The slicing criterion is the `vendingEnabled` instance variable at execution position 31 and 30. The differences between the two slices are indicated in

bold. `vendingEnabled` was selected as the slicing criterion because it is an instance variable essential to the class. It controls whether or not vending should be allowed. This approach, like the previous one, would have uncovered the fault in the modified version. The method call `coinBox.returnQtrs()` is not part of its slice, while it should have been, as `returnQtrs()` is the method responsible for resetting `vendingEnabled` to false.

The main benefit of the coarse-grained slicing based testing approach is that it reduces the amount of information provided. It preserves only the parts of the program's behavior related to the selected variable. This in turn narrows down the search space for the localization of a possible run time behavioral fault. However, the limitation with this approach lies in the fact that sometimes, in the case of a subtle error in the modified version, the slice computed for both programs can be the same and therefore, the fault remains undetected. To overcome such difficulties, the last approach is proposed.

3.4 Fine-Grained Slicing Based Testing

The fine-grained slicing based testing approach is a refinement of the previous one. Like its predecessor, it consists of computing a dynamic slice for the baseline and modified versions of a program. However, in order to further improve the understanding of the program's behavior, the list of contributing actions and influencing variables are identified at each step of the program's execution, as indicated in Figures 3.4.1 and 3.4.2. The terms action, contributing action, and influencing variable are defined as follows:

ACTION. During the execution of a program, a statement can be executed several times. An action corresponds to the particular execution of a program statement. More formally, action Y^t represents the program statement Y at position t in the execution trace T_x [KOR97B]. In Figures 3.4.1 and 3.4.2, actions are indicated in the left part of the first column.

CONTRIBUTING ACTION. Y^t is a contributing action with respect to the slicing criterion v^q, i.e., variable v at position q, if its execution has an effect on the computation of v^q [RIL98]. In Figures 3.4.1 and 3.4.2, contributing actions are highlighted in bold.

INFLUENCING VARIABLE. z^p is an influencing variable with respect to the slicing criterion v^q if (1) there exists a contributing action Y^t between positions p and q ($p < t < q$), (2) Y^t uses the value of z, and (3) z is not modified between p and t [KOR87, RIL98]. In Figures 3.4.1 and 3.4.2, influencing variables are indicated before the execution of each action.

Contributing Actions	Influencing Variables
44[1] public static void main(String args[])	
45[2] CoinBox coinBox = new CoinBox();	
7[3] public CoinBox()	coinBox
8[4] qtrsCollected = 0;	coinBox
9[5] qtrsInserted = 0;	coinBox
10[6] vendingEnabled = false;	coinBox
11[7] printTotal();	coinBox
38[8] public void printTotal()	coinBox
39[9] System.out.println((float)qtrsInserted...	coinBox
46[10] coinBox.insertQtr();	coinBox
14[11] public void insertQtr()	coinBox
15[12] qtrsInserted++;	coinBox
17[13] if (qtrsInserted > 1)	coinBox
20[14] printTotal();	coinBox
38[15] public void printTotal()	coinBox
39[16] System.out.println((float)qtrsInserted...	coinBox
47[17] coinBox.insertQtr();	coinBox
14[18] public void insertQtr()	coinBox

Figure 3.4.1 Contributing Actions and Influencing Variables for the Baseline Version

Contributing Actions	Influencing Variables
15^{19} qtrsInserted++;	coinBox
17^{20} if (qtrsInserted > 1)	coinBox
18^{21} vendingEnabled = true;	coinBox
20^{22} printTotal();	coinBox
38^{23} public void printTotal()	coinBox
39^{24} System.out.println((float)qtrsInserted...	coinBox
48^{25} coinBox.returnQtrs();	coinBox
23^{26} public void returnQtrs()	coinBox
24^{27} qtrsInserted = 0;	coinBox
25^{28} vendingEnabled = false;	coinBox
26^{29} printTotal();	
38^{30} public void printTotal()	
39^{31} System.out.println((float)qtrsInserted...	

Figure 3.4.1 Contributing Actions and Influencing Variables for the Baseline Version (Continued)

Contributing Actions	Influencing Variables
44^{1} public static void main(String args[])	
45^{2} CoinBox coinBox = new CoinBox();	
7^{3} public CoinBox()	coinBox
8^{4} qtrsCollected = 0;	coinBox
9^{5} qtrsInserted = 0;	coinBox
10^{6} vendingEnabled = false;	coinBox, qtrsInserted
11^{7} printTotal();	coinBox, qtrsInserted
38^{8} public void printTotal()	coinBox, qtrsInserted
39^{9} System.out.println((float)qtrsInserted...	coinBox, qtrsInserted
46^{10} coinBox.insertQtr();	coinBox, qtrsInserted
14^{11} public void insertQtr()	coinBox, qtrsInserted
15^{12} qtrsInserted++;	coinBox, qtrsInserted
17^{13} if (qtrsInserted > 1)	coinBox, qtrsInserted
20^{14} printTotal();	coinBox, qtrsInserted
38^{15} public void printTotal()	coinBox, qtrsInserted
39^{16} System.out.println((float)qtrsInserted...	coinBox, qtrsInserted
47^{17} coinBox.insertQtr();	coinBox, qtrsInserted
14^{18} public void insertQtr()	coinBox, qtrsInserted
15^{19} qtrsInserted++;	coinBox, qtrsInserted
17^{20} if (qtrsInserted > 1)	coinBox, qtrsInserted
18^{21} vendingEnabled = true;	coinBox
20^{22} printTotal();	
38^{23} public void printTotal()	
39^{24} System.out.println((float)qtrsInserted...	
48^{25} coinBox.returnQtrs();	
23^{26} public void returnQtrs()	
24^{27} qtrsInserted = 0;	
26^{28} printTotal();	
38^{29} public void printTotal()	
39^{30} System.out.println((float)qtrsInserted...	

Figure 3.4.2 Contributing Actions and Influencing Variables for the Modified Version

The comparison of the influencing variables and contributing actions determines if the run time behavior of the modified version is different from the baseline one. In the present example, a dynamic slice was computed using the same criterion as in the previous approach. In the modified version of the `CoinBox` class, `qtrsInserted` is an

influencing variable while it is not the case in the baseline version. This would have led to the early localization of the fault in the modified version. `qtrsInserted` cannot influence the computation of the `vendingEnabled` instance variable, since method `returnQtrs()` was invoked.

This approach can also be used to analyze the performance of both versions of a program, as it highlights the parts of the execution trace which are not relevant to the computation of the slicing criterion.

4. IMPLEMENTATION

4.1 CONCEPT

The three testing approaches proposed in the present research were implemented as part of the CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) research project [RIL02].

The major goal of the CONCEPT project is to address the current and future challenges related to the comprehension of large and distributed systems at the source code and architectural levels. Its objective is to provide software developers with novel comprehension approaches based on different source code analysis, visualization, reverse engineering, and architectural recovery techniques as well as their applications.

4.2 CONCEPT ARCHITECTURE

A lightweight reverse engineering environment supporting the different approaches presented within the context of the CONCEPT project was implemented. This was done to demonstrate how they can assist software developers regarding particular source code comprehension applications. Its architecture is illustrated in Figure 4.2.1.

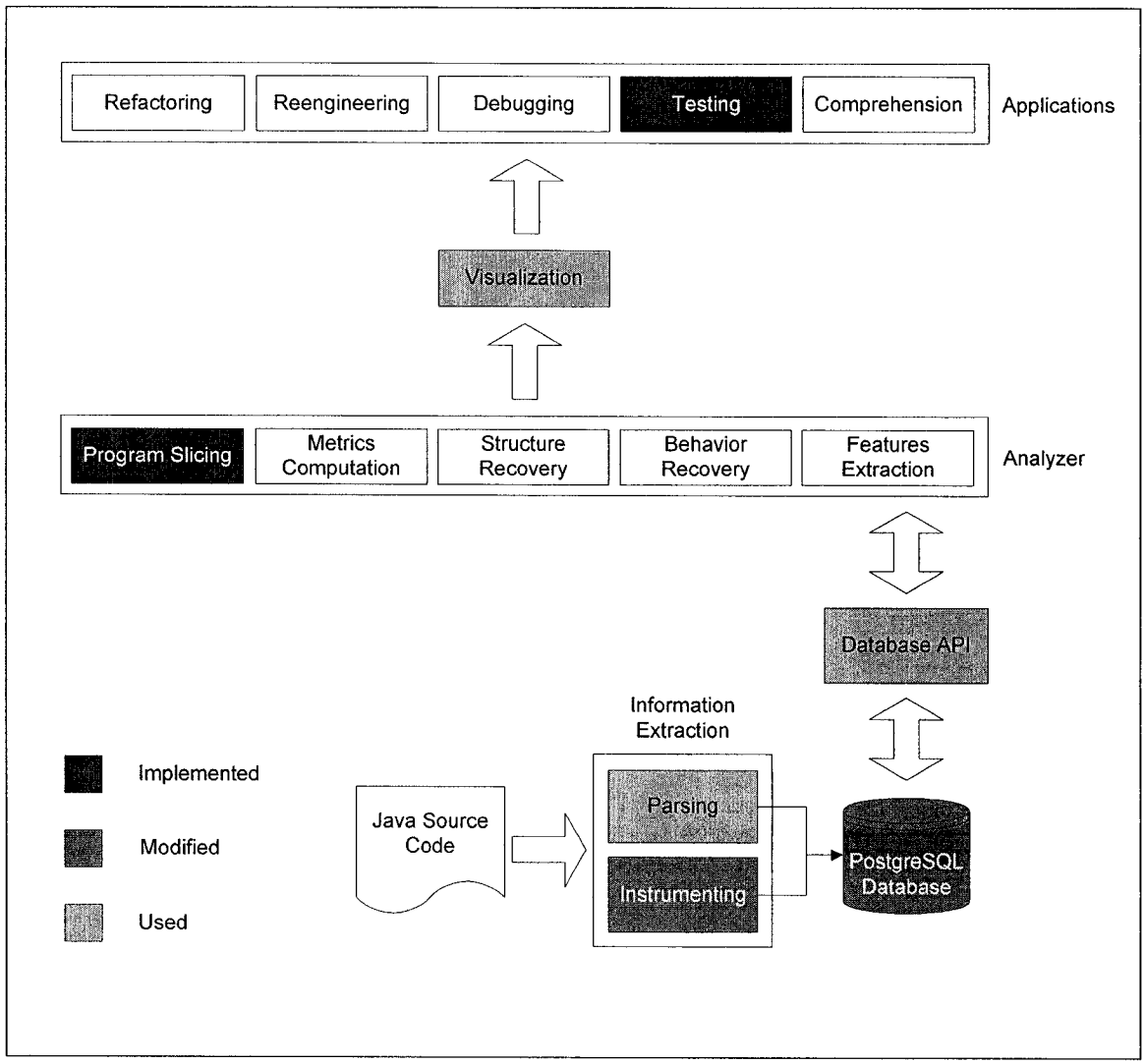

Figure 4.2.1 CONCEPT Architecture

The CONCEPT environment was implemented using a layered system architecture, with the PostgreSQL database acting as a repository for all shared data. The components of the diagram which are in black were implemented as part of this thesis. The ones in dark gray already existed. They were modified to suit the needs of the present research. The components in light gray are used by the implemented and modified ones.

The database stores information extracted by parsing and instrumenting the source code of Java programs. This information can be accessed by the different components of the Analysis layer using the Database API. The results generated by the analysis of a program of interest can then be visualized using the Visualization layer and used by the different applications.

4.3 PARSING

A necessary first step in supporting the comprehension of a system is the extraction of a concrete model, i.e., a representation of the implemented system. This model contains, among other things, a set of elements (e.g., files, classes, methods, variables), a set of relations between the elements (e.g., a file contains classes, a class contains methods, a method defines variables), as well as a set of attributes of these elements and relations (e.g., method *A* calls method *B* *n* times, variable *C* has type *D*) [KAZ99]. A model can represent the different views of a system. A view can be classified as either static or dynamic. A static view is obtained by observing only the artifacts of the system, while a dynamic view is acquired by observing the system during its execution [KAZ03].

In the CONCEPT environment, the static view of a system is extracted using a parser. There exists several parsers and parser generators for the Java programming language. The one which was selected is javac. It was preferred over the other ones because it can perform semantic analysis and its source code is available. In [ZHA03], Yonggang

Zhang, a member of the CONCEPT project, modified it to suit the needs of the research group. Its structure is illustrated in Figure 4.3.1.

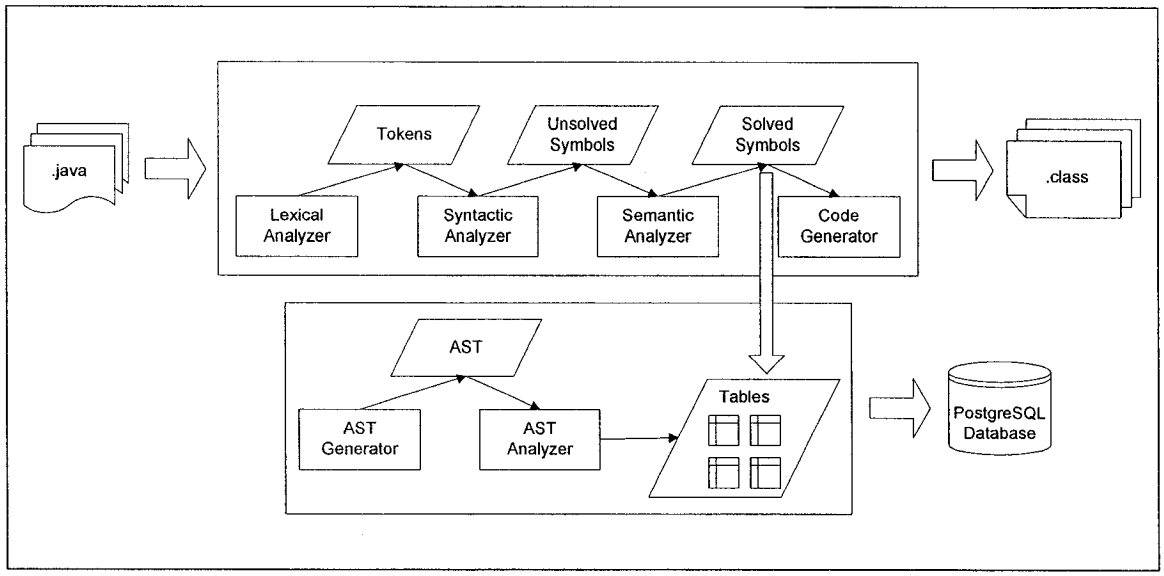

Figure 4.3.1 Parser Structure

The modified javac behaves just like the javac compiler included in the Java 2 Standard Developer's Kit (SDK): it reads class and interface definitions and compiles them into bytecode class files. After the symbol table is created, another program is invoked. This program reads the symbol table to generate a predefined Abstract Syntax Tree (AST) structure for each of the compiled classes. These ASTs are then partially normalized and stored in their corresponding table in the PostgreSQL database. Figure 4.3.2 displays the partial AST of a method.

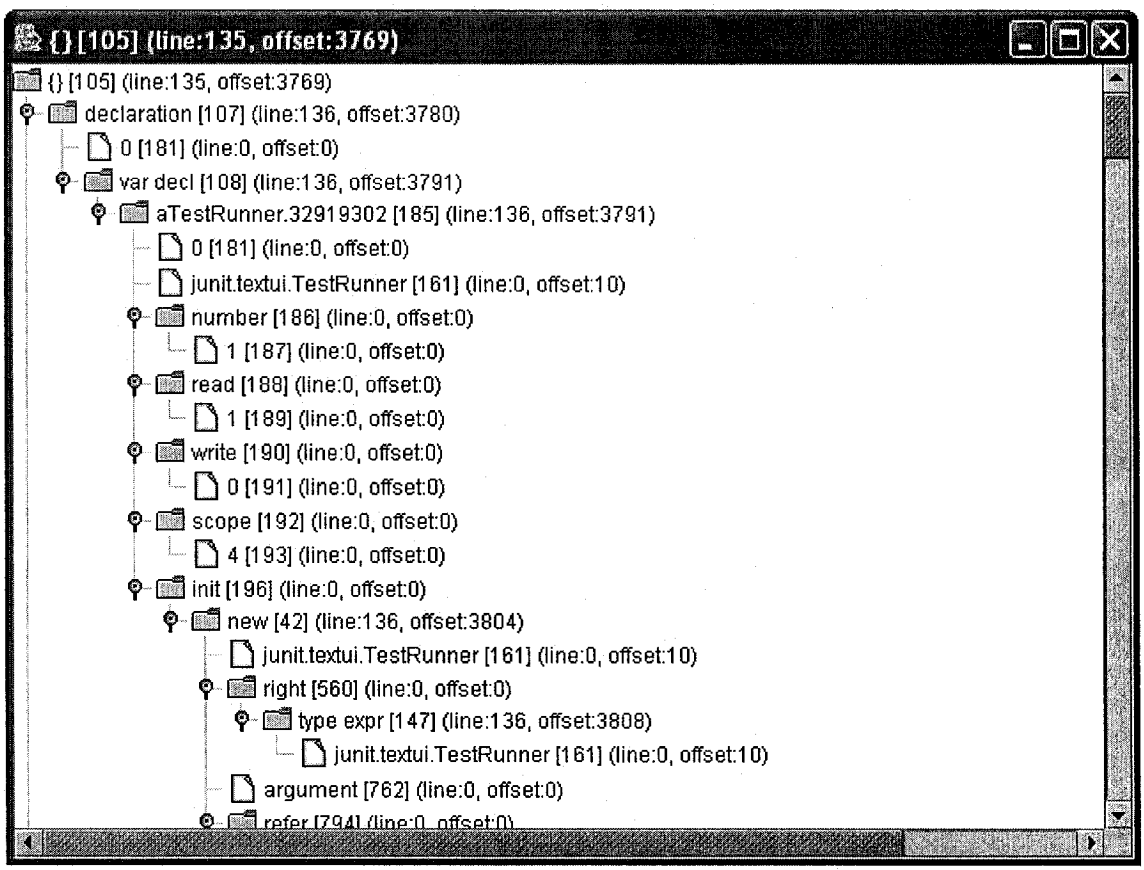

Figure 4.3.2 AST Example

4.4 DATABASE API

It is very difficult to analyze a program using the semantic level ASTs generated by the parser. A large number of recursive operations are needed to traverse an AST and search for a particular piece of information. For this reason, each AST is further parsed into an object-oriented model. This reduces the complexity of extracting the information from the PostgreSQL database and acts as an API used to analyze the results generated by the javac parser. Figure 4.3.3 shows the partial class diagram of the database API [ZHA03].

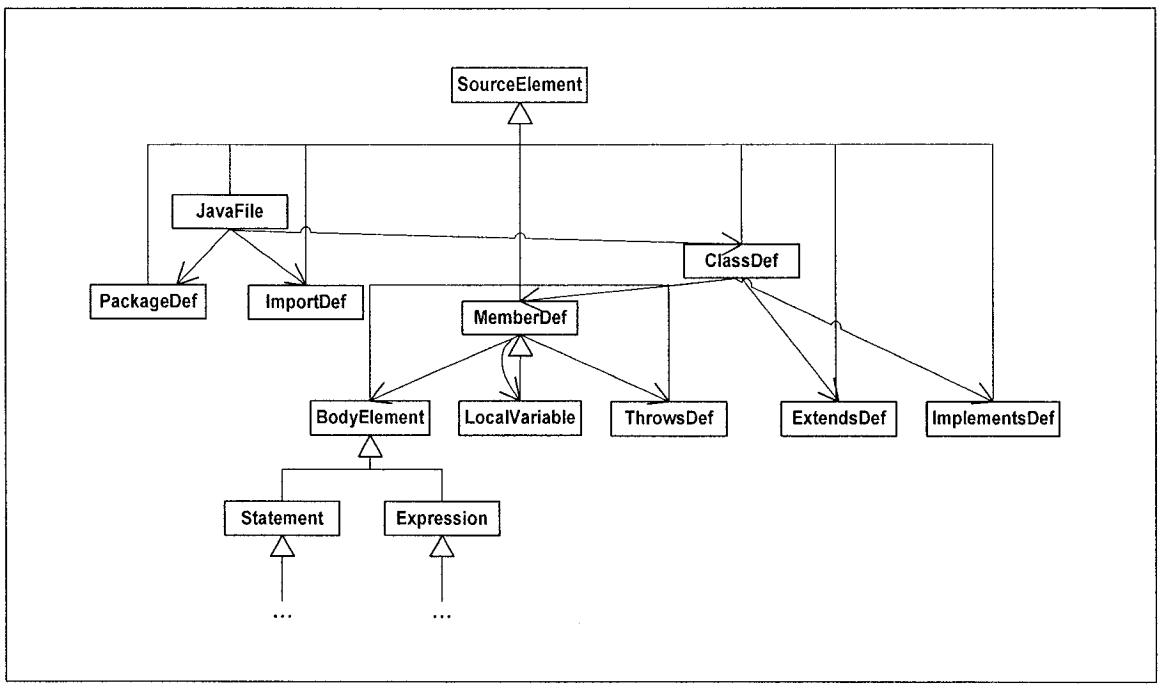

Figure 4.3.3 Partial Class Diagram of the Database API

As illustrated in Figure 4.3.3, each element of the Java programming language is either represented as an object of type `SourceElement` or as one of its subclasses. The class `JavaFile` represents a java source code file which contains a package statement (`PackageDef`), an import statement (`ImportDef`), and a class definition (`ClassDef`). A class definition can inherit another class (`ExtendsDef`) and implement an interface (`ImplementsDef`). The class `MemberDef` represents an instance variable, a method, or an inner class, in which statements and expressions can be defined.

4.5 Extraction of Run Time Information

In [LOW01], Löwe et al. show that there are four different alternatives to obtain run time information: Code instrumentation, annotation of run time environments, post mortem

analysis, and on-line debugging or profiling. The two alternatives which were considered in the present research are the on-line debugging approach using the Java Debug Interface and automatically instrumenting the source code using an instrumentation toolkit.

4.5.1 On-Line Debugging Using the Java Debug Interface

The Java Debug Interface is part of the Java Platform Debugger Architecture (JPDA), a multi-tiered debugging architecture which allows tool developers to easily create debugger applications that run portably across platforms, virtual machine implementations, and software development kit versions [SUN04]. The JPDA consists of three layers: the Java Virtual Machine Debug Interface (JVMDI), the Java Debug Wire Protocol (JDWP), and the Java Debug Interface (JDI).

The JVMDI is the lowest layer within the JPDA. It defines the services a virtual machine must provide for debugging, such as inspecting the state of a running Java application and controlling its execution. The JDWP is the protocol used by a debugger and the virtual machine it debugs to communicate. It defines the format of information and requests transferred between them. The JDI is the highest layer of the JPDA. It is an application programming interface that provides useful information for debuggers and similar systems, which need access to a running virtual machine's state as well as explicit control over its execution. Even though implementers of debuggers can use the JDWP or JVMDI directly, the JDI makes easier the integration of debugging capabilities into

development environments. Therefore, Sun Microsystems suggests its use for the development of debugger applications.

Using the JDI, an application was developed to run a Java program and generate a trace of its execution. One advantage of this approach is that it does not modify the source code of the program to trace. It also provides an introspective access to a running virtual machine's state, classes, arrays, interfaces, primitive types, and instances of those types. Furthermore, it allows to have explicit control over a virtual machine's execution, such as the possibility to suspend and resume threads, set breakpoints, watchpoints as well as inspect a suspended thread's state, local variables, and stack backtrace [SUN04].

However, this approach has some severe performance problems [HEU02]. These are caused by the fact that the program which is being debugged launches its own virtual machine. This results into expensive interprocess communication between the debugger and debuggee processes.

For example, starting the SwingSet2 application available on the Java 2 SDK version 1.4.2, loading all the demos it contains, and then closing it takes approximately 4.13 seconds on a 2.66 GHz Pentium 4 with 1024 MB of memory running Windows XP. In comparison, when the tracing program with the JDI is used to run the same application and collect information related to this particular program execution, the whole process takes about 108.38 seconds, a 2524.21% increase.

	Number of Executed Statements	Time in Seconds
SwingSet2	17 823	4.13
SwingSet2 + JDI	17 823	108.38

Table 4.5.1.1 Overhead of the On-Line Debugging Approach Using the JDI

4.5.2 Automatically Instrumenting the Source Code

Due to the performance issue outlined above, another approach was considered. It consists of instrumenting the source code of a Java program in order to trace each source code line as it is executed.

The instrumentation of the source code is done automatically using the instrumentation toolkit developed by Glen McCluskey & Associates LLC [GLE04], an object-oriented technology firm based in Colorado and specializing in the C++ and Java programming languages.

The toolkit consists of the instr and query Java packages. instr instruments Java source code. It can be used to count the number of times each statement of a program is executed, do method-level instrumentation, or trace individual source code lines. instr is built on the query package, which parses a Java source program into an internal tree structure. instr then operates on the tree representation of the program, adding additional information to it.

One of the programs supplied with the instr package is instr_trace. It allows to instrument a Java program such that each of its source code line is printed on the console

output as it is executed. instr_trace accomplishes this by performing the following steps for each of the program files. First, it reads the source code file and parses it into a tree structure while preserving all the comments and white spaces. The parse tree is subsequently annotated with the necessary instrumentation. Afterwards, the annotated tree is written back to the file. To collect the run time information, the instrumented file then needs to be compiled and executed. The instrumentation code can later be removed using another program called uninstr.

Figures 4.5.2.2 and 4.5.2.3, taken from the instr documentation, show respectively an excerpt from the instrumented program of Figure 4.5.2.1 and the output it generates. The `/*_I*/` and `/*I_*/` are structured comments which are used to enclose the added instrumentation code.

```
1 public class loop {
2   public static void main(String args[])
3   {
4     int n = 1;
5     for (int i = 1; i <= 10; i++)
6       n = n * i;
7     System.out.println(n);
8   }
9 }
```

Figure 4.5.2.1 Sample Program

```
public class loop {
  public static void main(String args[])
    {/*_I*/instr.InstrUtil.showLine("loop.java", 2);/*I_*/
}
```

Figure 4.5.2.2 Excerpt of the Instrumented Sample Program

```
[loop.java 2]   public static void main(String args[])
[loop.java 4]     int n = 1;
[loop.java 5]     for (int i = 1; i <= 10; i++)
[loop.java 6]       n = n * i;
[loop.java 6]       n = n * i;
[loop.java 6]       n = n * i;
[loop.java 6]       n = n * i;
[loop.java 6]       n = n * i;
[loop.java 6]       n = n * i;
[loop.java 6]       n = n * i;
[loop.java 6]       n = n * i;
[loop.java 6]       n = n * i;
[loop.java 6]       n = n * i;
[loop.java 7]     System.out.println(n);
3628800
```

Figure 4.5.2.3 Output Generated by the Instrumented Sample Program

This approach reduces the overhead of the previous one. The time it takes to start the instrumented SwingSet2 application is approximately 7.98s, a 92.64% improvement.

	Number of Executed Statements	Time in Seconds
SwingSet2	17 823	4.13
SwingSet2 + instr	17 823	7.98
SwingSet2 + JDI	17 823	108.38

Table 4.5.2.1 Overhead of Instrumenting the Source Code Using instr

4.5.3 Modifications Made to instr

One problem with the instr_trace program is that its output is written on the console. Furthermore, even if it is redirected from the console output to a file, it tends to produce files of a considerable size for large execution traces, since for each executed source code line, the file name, line number, and source code are written. For this reason, the instr_trace program was decompiled using the JAD decompiling engine [JAD04] to reconstruct the original source code from the compiled binary class files. instr_trace was then modified in order to write the execution trace directly to a file. Moreover, to reduce

the size of the output file, only a key uniquely identifying the file name and the line number are written for every executed program source code line. The file key is automatically generated when the Java program to analyze is being parsed.

Another modification made to the instr package consisted of adding a program which reads an output file generated by instr_trace and stores it into the PostgreSQL database. This modification also required the addition of two tables in the database to store the run time information.

4.6 PROGRAM SLICING ALGORITHM

The program slicing algorithm used in the coarse and fine-grained slicing based testing approaches is a modified version of the dynamic program slicing algorithm with removable blocks proposed by Korel [KOR95, KOR97A].

4.6.1 Removable Block

Korel defines the concept of removable block (or block) as "the smallest part of program text that can be removed during slice computation without violating the syntactical correctness of the program" [KOR97A]. Examples of blocks are assignment, input, and output statements. The conditional expressions of control statements are not removable and as a result, are not considered as blocks. Each block *B* has a regular entry and exit,

referred respectively as its r-entry and r-exit. In Figure 4.6.1.1, the removable blocks of a sample program are displayed as rectangles.

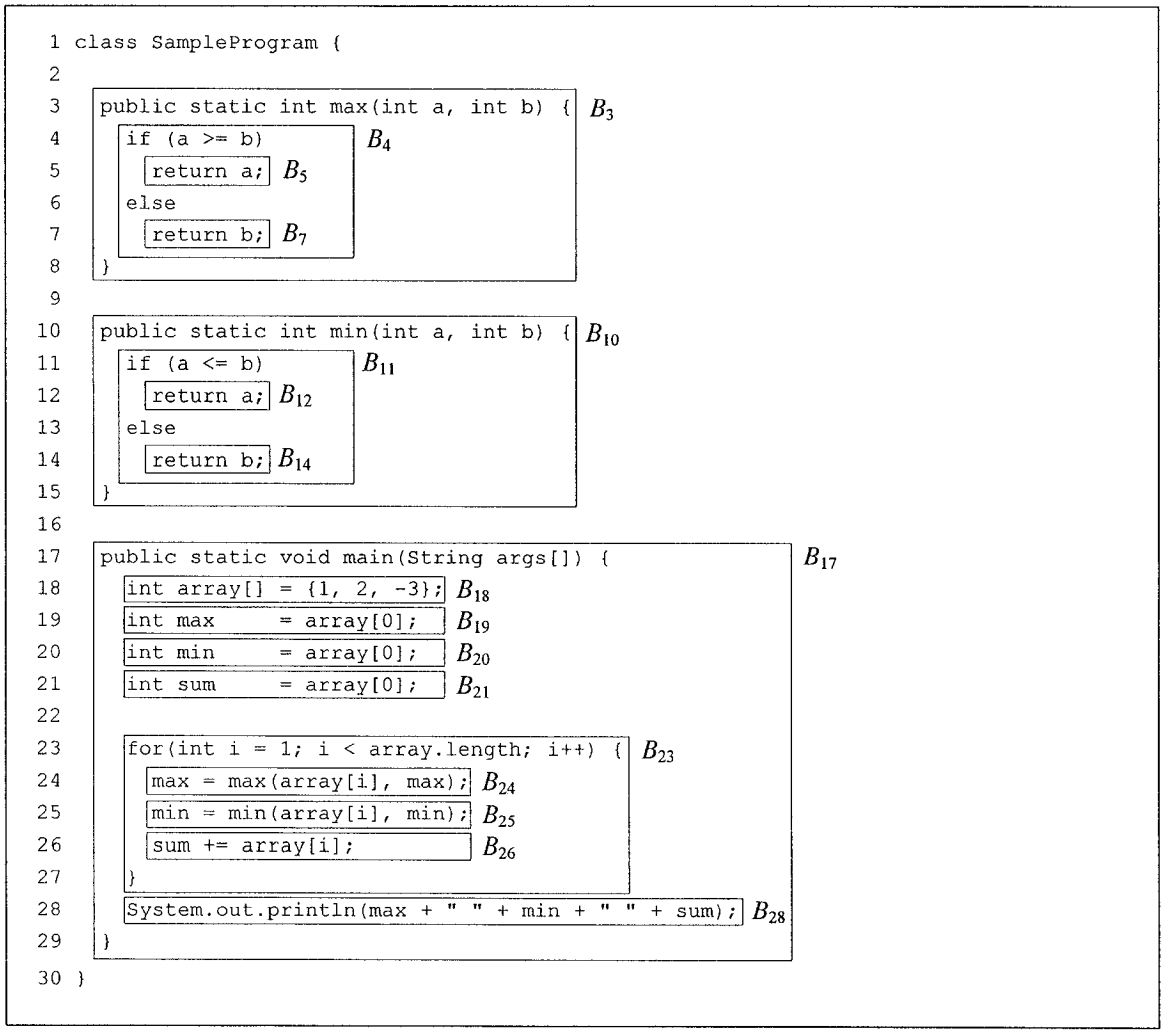

Figure 4.6.1.1 Removable Blocks of a Sample Program

4.6.2 Dynamic Program Slicing Algorithm with Removable Blocks

The goal of the other dynamic program slicing algorithms, e.g., [AGR90, GOP91, and KOR88], is to identify the actions in the execution trace T_x which contribute to the computation of a slicing criterion $C = (x, y^q)$. This is achieved by deriving the data and

control dependencies. However, identifying the actions which do not contribute to the computation of variable y^q is equally important. The more non-contributing actions are identified, the smaller may be the resulting slice.

In the present algorithm, the data dependencies are used to identify the contributing actions and the removable blocks, the non-contributing ones. Informally, a block can be removed from a program if its removal does not disorder the flow of execution for input x and none of the executed actions it contains contribute to the computation of y^q. Let B_1, B_2, and B_3 be a sequence of three blocks. Block B_2 can be removed if during the execution of the program for input x (1) the execution exits from block B_1 through its r-exit, (2) enters block B_2 through its r-entry, (3) leaves B_2 through its r-exit, (4) enters block B_3 through its r-entry, and (5) none of the executed actions within B_2 contribute to the computation of y^q [KOR97A]. If B_2 is removed and the program is executed for the same input x, then after leaving B_1 through its r-exit, the execution will enter B_3 through its r-entry. This removal will not affect the flow of execution nor the computation of variable y^q.

4.6.3 Description of the Algorithm

The dynamic program slicing algorithm with removable blocks is displayed in Figure 4.6.3.2 [RIL98]. The following concepts are used in the algorithm. $U(Y^p)$ is the set of variables used at action Y^p. $D(Y^p)$ is the set of variables defined at action Y^p. The last

definition of variable v^k in the execution trace T_x is the action Y^p such that (1) $v \in D(Y^p)$ and (2) for all i, $p < i < k$ and all Z such that $T_x(i) = Z$, $v \notin D(Z^i)$ [KOR97A].

N(B) designates all the program statements contained within block *B*. *S(B, k_1, k_2)* denotes a block trace, which is the part of the execution trace corresponding to the execution of block *B*. More formally, a block trace is a subtrace of execution trace T_x where (1) k_1 is the position of the r-entry of block *B*, (2) k_2 is the position of the r-exit of *B*, and (3) the execution does not exit from block *B* through its r-exit between k_1 and k_2 - 1 [KOR97A].

Figure 4.6.3.1 shows the execution trace as well as the block traces of the sample program displayed in Figure 4.6.1.1.

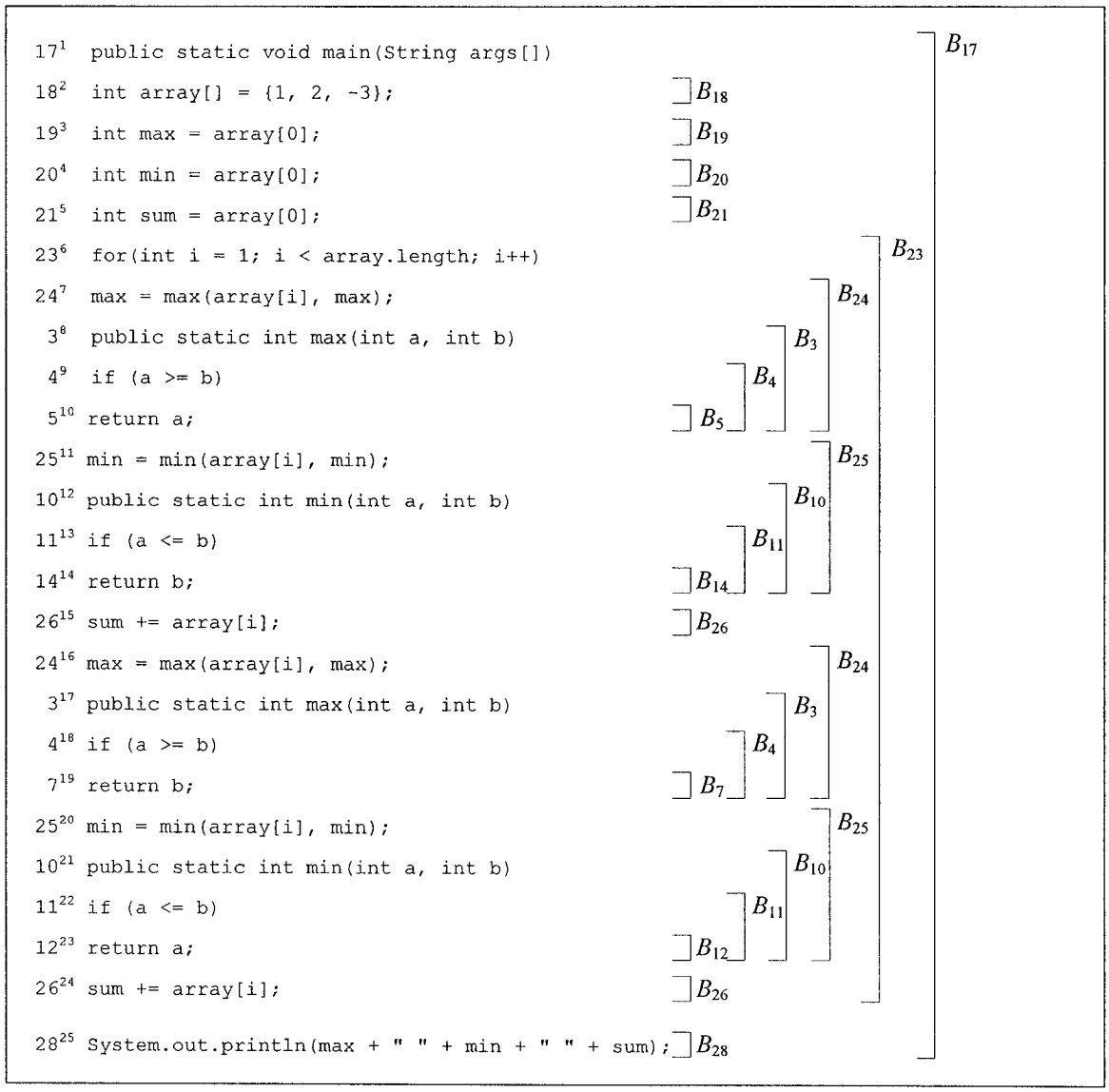

Figure 4.6.3.1 Execution Trace and Block Traces of the Sample Program

Input: a slicing criterion $C = (x, y^q)$
Output: a dynamic slice for C

T_x: execution trace up to position q
Φ_C: a set of block traces
R_C: a set of blocks
I_C: a set of contributing actions

1 Execute program P on input x and record execution trace T_x up to position q
2 Initialize R_C to a set of all blocks in program P
3 Mark all actions in T_x as neutral and not visited ($I_C := \emptyset$)
4 Find and mark as contributing the last definition of y^q
5 **repeat**
6 Find contributing actions
7 Find non-contributing actions
8 Mark all neutral actions as contributing
9 **until** all actions are marked as contributing or non-contributing in T_x up to position q
10 Show a dynamic slice that is constructed from P by removing all blocks that belong to R_C.

procedure Find contributing actions
11 **while** there exists a contributing and not visited action in T_x **do**
12 Select a contributing and not visited action X^k in T_x
13 Mark X^k as a visited action ($I_C := I_C \cup \{X^k\}$)
14 **for** all variables $v \in U(X^k)$ **do**
15 Find and mark as a contributing action the last definition of v
16 **endfor**
17 **for** all blocks $B \in R_C$ **do**
18 if $X \in N(B)$ **then** $R_C := R_C - \{B\}$
19 **endfor**
20 **endwhile**
end Find contributing actions

Figure 4.6.3.2 Dynamic Program Slicing Algorithm with Removable Blocks

```
    procedure Find non-contributing actions
21    Mark as neutral all actions that are not marked as contributing (Φ_C := Ø)
22    p := 1
23    repeat
24      Let X^p be an action at position p in T_x
25      if X^p is not a contributing action (X^p ∉ I_C) then
26        Let B be a block which has an r-entry at position p
27        if B ∈ R_C then
28          if there exists an r-exit from block B at position p_l such that
29            (1) p ≤ p_l ≤ q
30            (2) all actions between p and p_l are not marked as contributing
31          then
32            Mark all actions between p and p_l as non-contributing (Φ_C := Φ_C ∪
              {S(B,p,p_l)})
33            p := p_l
34          endif
35        endif
36      endif
37      p := p + 1
    end Find non-contributing actions
```

Figure 4.6.3.2 Dynamic Program Slicing Algorithm with Removable Blocks (Continued)

The first step of the algorithm consists of recording the execution trace of the program up to execution position q. R_C is then set to contain all the blocks in the program. Each action in the execution trace can be in one of the following states: contributing, non-contributing, or neutral. Initially, all actions are marked as neutral and not-visited. In step 4, the last definition of y^q is located and this action is marked as contributing.

The algorithm then iterates in the repeat loop consisting of lines 5 to 9 until all actions of the execution trace are either marked as contributing or non-contributing. Inside this loop, there are three steps. Step 6 identifies the actions which contribute to the computation of y^q. In step 7, using the set of contributing actions, the algorithm identifies the non-contributing ones. The actions which are not identified as contributing or non-

contributing are marked as contributing in step 8. They will be visited on the next iteration of the repeat loop.

In step 6, the contributing actions are identified. This procedure, which consists of a while loop, is detailed in lines 11 to 20. On each iteration, a contributing and not visited action X^k is selected, marked as contributing, and added to the set I_C of contributing actions. In lines 14 to 15, the last definition of each variable used at X^k is identified and marked as contributing. The next step consists of removing from R_C all the blocks which contain statement X. The while loop iterates until all the contributing actions have been visited.

In step 7, the non-contributing actions are identified. This procedure is presented in more detail in Figure 4.6.3.2. Initially, it marks all actions as neutral if they have not already been marked as contributing. It then searches the execution trace from the beginning looking for actions which have been marked as neutral. If such an action X^p is found, the procedure then tries to find a block trace $S(B, p, p_1)$ where B is in R_C and all the actions between p and p_1 are not marked as contributing. If such a block trace is found, then all the actions within this block trace are marked as non-contributing, the block trace is added to the set Φ_C, and the algorithm resumes its search at position p_1. If not, then the algorithm tries to find the next neutral position starting from position $p + 1$. This procedure continues until it reaches position q in the execution trace.

If the above algorithm is applied for variable `sum` at position 25 of the execution trace displayed in Figure 4.6.3.1, the slice is computed as follows:

After the first iteration of the repeat loop:

$I_C = \{18^2, 21^5, 23^6, 26^{15}, 26^{24}\}$

$R_C = \{B_3, B_4, B_5, B_7, B_{10}, B_{11}, B_{12}, B_{14}, B_{19}, B_{20}, B_{24}, B_{25}\}$

$\Phi_C = \{S(B_{19},3,4), S(B_{20},4,5), S(B_{24},7,11), S(B_{25},11,15), S(B_{24},16,20), S(B_{25},20,25)\}$

Action 17^1 is marked as contributing in step 8, since it does not belong to I_C nor to any of the block traces of Φ_C.

After the second iteration of the repeat loop:

$I_C = \{17^1, 18^2, 21^5, 23^6, 26^{15}, 26^{24}\}$

$R_C = \{B_3, B_4, B_5, B_7, B_{10}, B_{11}, B_{12}, B_{14}, B_{19}, B_{20}, B_{24}, B_{25}\}$

$\Phi_C = \emptyset$

The dynamic slice shown in Figure 4.6.3.3 is obtained by removing all the blocks which belong to R_C.

```
 1 class SampleProgram {
 2
17   public static void main(String args[]) {
18     int array[] = {1, 2, -3};
21     int sum     = array[0];
22
23     for(int i = 1; i < array.length; i++) {
26       sum += array[i];
27     }
28     System.out.println(max + " " + min + " " + sum);
29   }
30 }
```

Figure 4.6.3.3 Dynamic Slice for Variable *sum* at Statement 28

4.6.4 Modifications Made to the Algorithm

Figure 4.6.3.2 constitutes the dynamic program slicing algorithm with removable blocks proposed by Korel. Its correctness has been proved in [KOR95, KOR97A].

However, the problem with this algorithm is that it was designed for the procedural version of the Pascal programming language. As a result, it had to be modified to suit the context of this research and compute dynamic slices for object-oriented programs written in Java.

The modifications are contained in Figure 4.6.4.1. This procedure was added so that the algorithm can handle constructor and method calls. Even with this addition, the algorithm still computes correctly dynamic slices. The set of removable blocks R_C fulfill the four conditions a dynamic slice must satisfy as identified and proved by Korel in [KOR97A]. Informally, these conditions are as follows: (1) each action in the execution trace is either contributing or non-contributing, but not both, (2) the last definition of y^q is in I_C, (3) for each action X^k which belongs to I_C, the last definition of all the variables

used at X^k are also included in I_C, and (4) if an action X^k is in I_C, then statement X cannot belong to a block which is in R_C.

```
    procedure Mark action as contributing
38    Let X^p be an action at position p in T_x
39    Let S(B, p, p_1) be a block trace with an r-entry at position p
40    Mark X^p as a contributing action
41    for all method calls at X^p do
42      if the return type of the method is void then
43        mark as contributing all the actions which belong to the body of the method
          between positions p and p_1
44      else
45        mark as contributing the return statement of the method located between
          positions p and p_1
46      endif
47    endfor
48    for all constructor calls at X^k do
49      mark as contributing all the actions which belong to the body of the constructor
        between positions p and p_1
50    endfor
    end Mark action as contributing
```

Figure 4.6.4.1 Procedure Added to the Algorithm

If an action is marked as contributing and this action contains a method call with a return type other than void, then the return statement of the method body is also marked as contributing. For example, Figure 4.6.4.3 shows the execution trace of the sample program of Figure 4.6.4.2. If the first action in the execution trace is marked as contributing, then the return statement associated with the method call at position 4 will also be marked as contributing.

```
…
10 int result = abs(-1);
…
15 public static int abs(int a) {
16   if (a >= 0)
17     return a;
18   else
19     return -a;
20 }
…
```

Figure 4.6.4.2 Sample Program with a Method Call

Execution Trace	Actions State
… 10^1 int result = abs(-1); 15^2 public static int abs(int a) 16^3 if (a >= 0) 19^4 return -a; …	contributing Contributing

Figure 4.6.4.3 Execution Trace of the Sample Program of Figure 4.6.4.2

In the case of an action with a constructor or method call with a void return type, then all actions within the body of the constructor or method are also marked as contributing. For example, Figure 4.6.4.5 displays the execution trace of the sample program of Figure 4.6.4.4. If the first action of the execution trace is marked as contributing, then all actions from position 2 to 5 will also be marked as contributing.

```
…
10 Box myBox = new Box(10, 20, 15);
…
15 public Box (double w, double h, double d) {
16   width  = w;
17   height = h;
18   depth  = d;
19 }
…
```

Figure 4.6.4.4 Sample Program with a Constructor Call

Execution Trace	Actions State
... 10^{1} Box myBox = new Box(10, 20, 15); 15^{2} Box (double w, double h, double d) 16^{3} width = w; 17^{4} height = h; 18^{5} depth = d; ...	contributing contributing contributing contributing contributing

Figure 4.6.4.5 Execution Trace of the Sample Program of Figure 4.6.4.4

4.6.5 Implementation of the Algorithm

The modified version of the dynamic program slicing algorithm with removable blocks was implemented as part of this research using the Java programming language. The principal classes and relationships of its implementation are displayed in Figure 4.6.5.1.

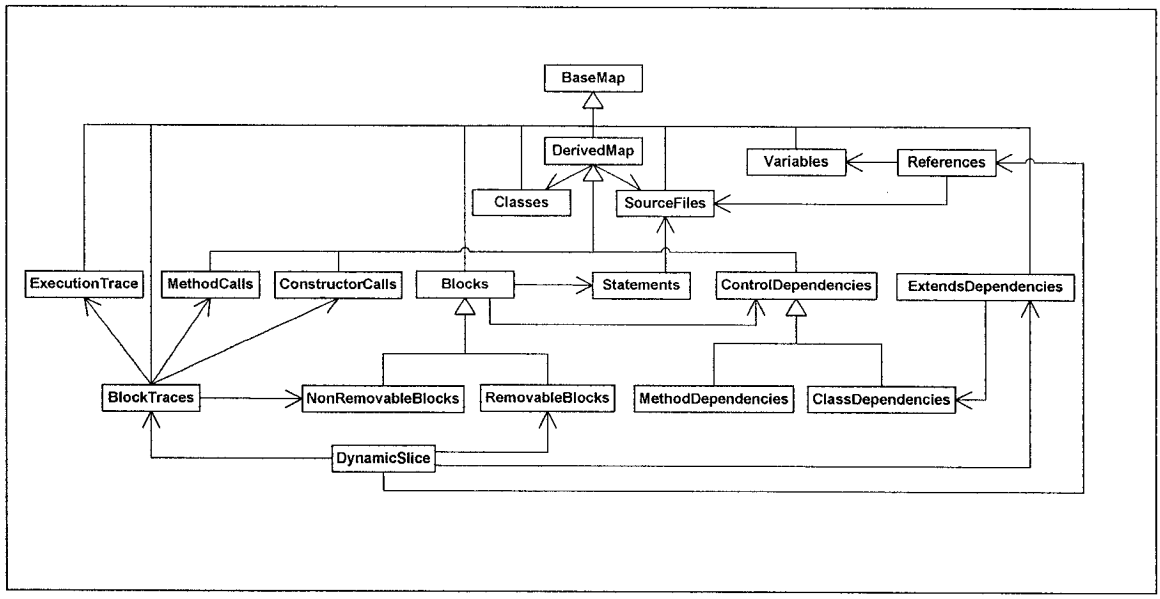

Figure 4.6.5.1 Class Diagram of the Dynamic Program Slicing Algorithm

The algorithm itself is implemented in the `DynamicSlice` class. However, in order to compute slices, it needs both static and dynamic information about the Java program of

interest. This information is contained in the remaining classes, most of which are subclasses of `BaseMap`. As indicated by its name, the `BaseMap` class stores associations between keys and values.

The dynamic information required by the algorithm is stored in the `ExecutionTrace` and `BlockTraces` classes. These represent respectively a recorded execution trace, as generated by the instr_trace program, and the corresponding set of block traces.

The remaining classes contain the static information which is generated by the parser. The information about the defined classes is stored in the class of the same name, while the data about the files which hold the source code of a program are stored in the `SourceFiles` class. Since both classes are used by several others, a subclass of `BaseMap` was defined which holds a reference for each of them.

The `MethodCalls` and `ConstructorCalls` classes contain respectively all the methods and constructors which are invoked throughout the program. These are used to generate the block traces as well as compute slices.

As indicated by their names, the class `RemovableBlocks` stores the blocks which can be removed by the program slicing algorithm, e.g., methods, constructors, and statements. On the other hand, the `NonRemovableBlocks` class stores the ones which cannot be removed, e.g., classes, instance variables, and interfaces. In order to generate the set of blocks of a program, control dependencies are needed. These are defined in the

`ControlDependencies` class. The `MethodDependencies` subclass contains the control dependencies inside methods, while the `ClassDependencies` subclass contains the ones for classes. The `MethodDependencies` are used to generate the set of removable blocks, and the `ClassDependencies`, the non-removable ones.

In order to compute slices, the algorithm needs the set of variables which are defined and used at each step of the program's execution. This information is stored in the `References` class. This class contains all the variables which are declared, written, and read at each source code line.

The `ExtendsDependencies` and `Statements` are the last two classes. The first one contains the inheritance hierarchies. The second one stores the location of a number of particular program statements, e.g., if, while, for, and return. These are used to generate the set of block traces and compute slices.

4.7 TESTING

4.7.1 Execution Based Testing

As mentioned in section 4.5.3, the execution trace generated by instr_trace is written into a file and consists of a file key and line number for each of the executed source code line. To compare and measure the differences between the execution traces of the baseline and

modified versions of a program, the Beyond Compare [SCO04] utility version 2.2.3 is used.

Beyond Compare, by Scooter Software, is an advanced file and folder comparison utility for Windows. It contains a wide range of file and text operations as well as script commands for automating tasks. Its major components are a side-by-side folder and file viewers, which allow to visualize the differences between two files or folders in detail and carefully reconcile them.

Once the execution traces of the baseline and modified versions of a program are recorded, the resulting files are opened with Beyond Compare. Figure 4.7.1.1 shows a screen capture of two execution traces that were analyzed. These are the ones of the `CoinBox` class of section 3.1.

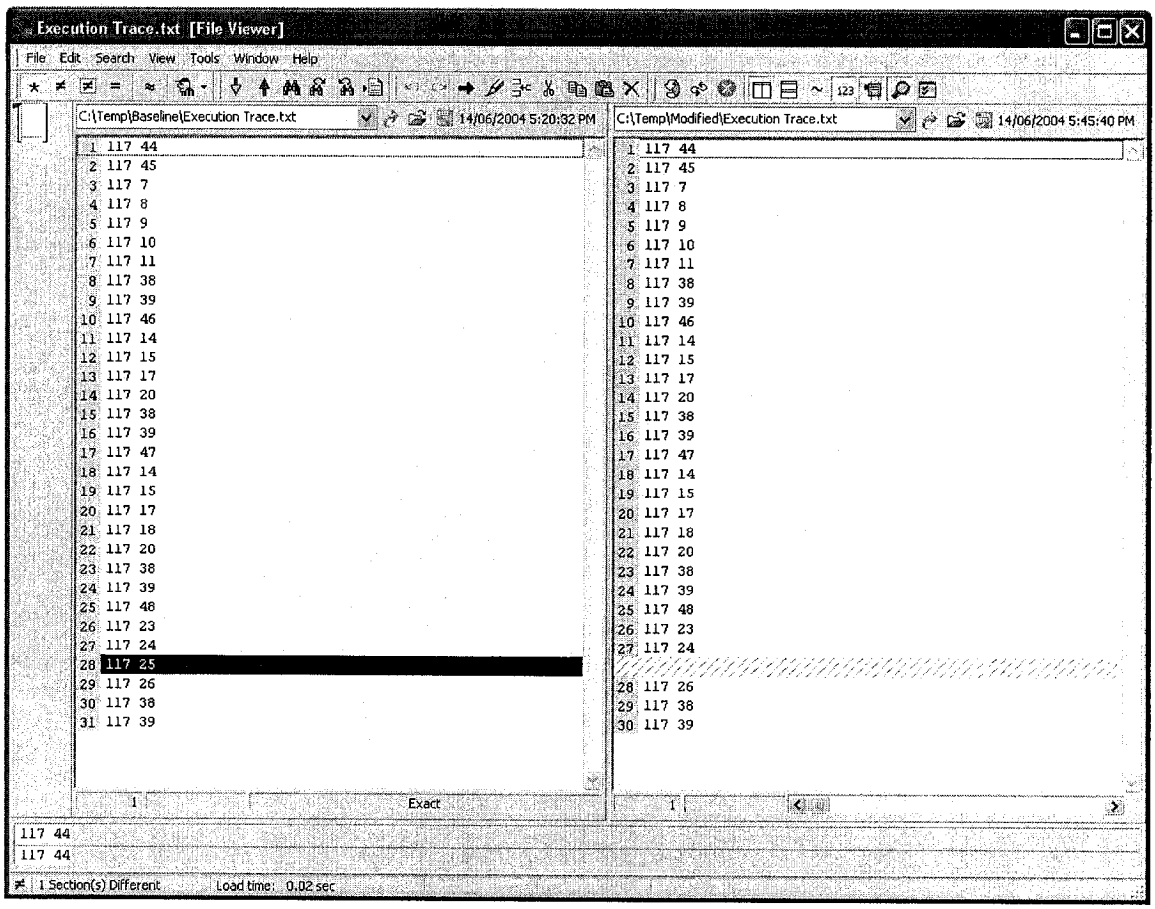

Figure 4.7.1.1 Comparison of Execution Traces in Beyond Compare

In the present example, Beyond Compare clearly highlights the difference between the two execution traces. As mentioned previously, this difference is due to the fact that one statement in the modified version of the `CoinBox` class was commented out and as a result, was not executed.

On the left edge of the display, there is a thumbnail view. It represents each line of the comparison as a colored line of one pixel high. It allows to see at a glance the pattern of differences throughout the comparison. The white rectangle shows the current view displayed in the main window, while the small triangle corresponds to the line currently

displayed. Clicking on a line of the thumbnail will display the line at that position in the main window. The thumbnail view is particularly useful when execution traces of a considerable size are compared.

4.7.2 Coarse-Grained Slicing Based Testing

The output of the program slicing algorithm of section 4.6.3 consists of a set of blocks which need to be removed from a program in order to obtain a dynamic slice. Like in the previous approach, the Beyond Compare utility is used to analyze the resulting slices for the baseline and modified versions of a program.

Once the non-contributing blocks have been removed from the two versions of the program, the complete paths containing the slices are typed into the left and right edit controls of Beyond Compare, as illustrated in Figure 4.7.2.1.

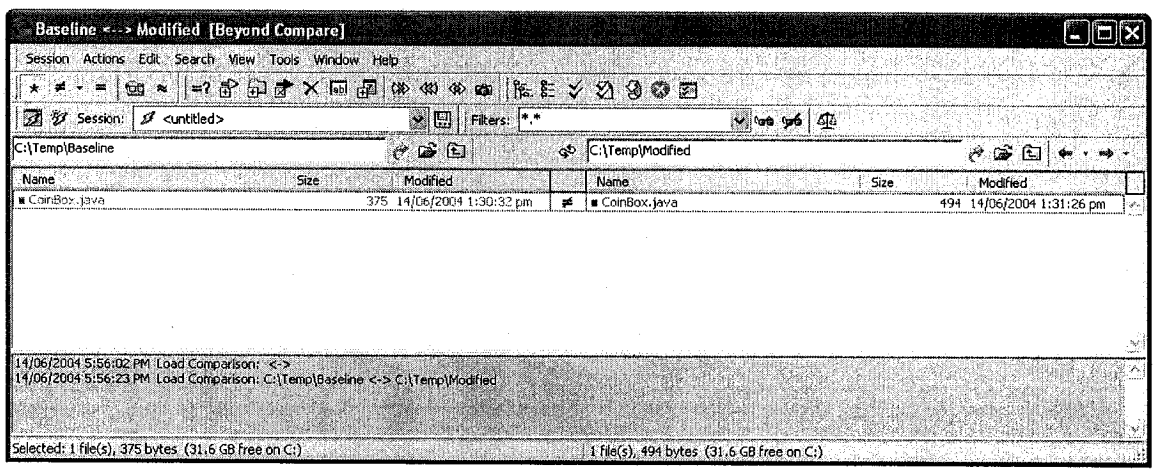

Figure 4.7.2.1 Comparison of Folders in Beyond Compare

Figure 4.7.2.1 shows that the files contained in the two folders are different. Clicking on the selected line will open another window which highlights the differences between the two dynamic slices. This is illustrated in Figure 4.7.2.2.

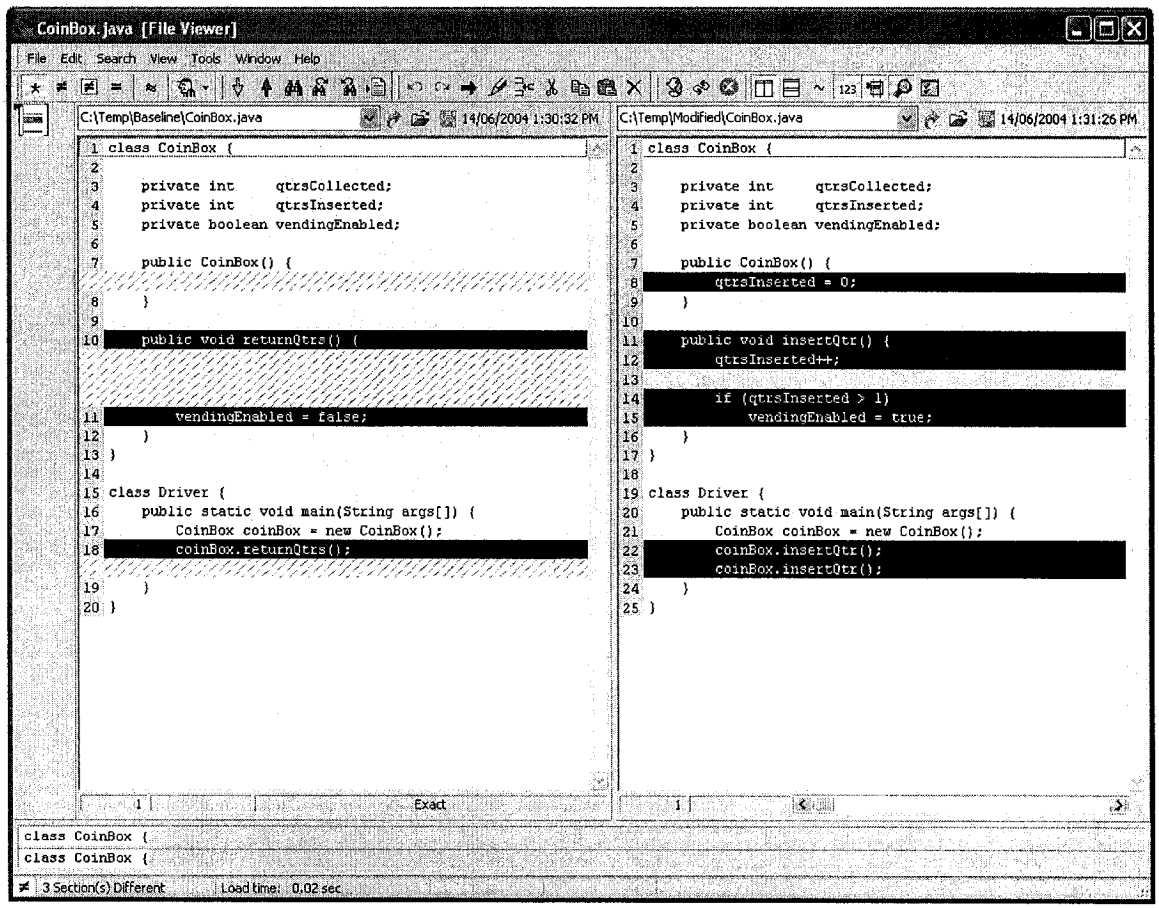

Figure 4.7.2.2 Comparison of Slices in Beyond Compare

Even though the two slices of Figure 4.7.2.2 consist of only one file, Beyond Compare is also able to differentiate slices consisting of several files stored in a number of directories.

4.7.3 Fine-Grained Slicing Based Testing

For the last testing approach, an algorithm was designed and implemented in Java to compute the set of influencing variables at each step of the program execution. This algorithm is detailed in Figure 4.7.3.1.

The algorithm takes as parameters a recorded execution trace T_x on input x, a slicing criterion $C = (x, y^q)$, and a set of contributing actions I_C. All of these parameters are the same as the ones used in the dynamic program slicing algorithm with removable blocks of section 4.6.3.

Input: an execution trace T_x on input x up to position q
a slicing criterion $C = (x, y^q)$
a set of contributing actions I_C
Output: an execution trace annotated with the set of influencing variables at each point of the program execution

```
1  Find the last definition X^p of y^q
2  for i := q to p + 1 do
3     add y to the set of influencing variables for action X^i
4  endfor
5  for all actions X^k ∈ I_C do
6     for all variables v ∈ U(X^k) do
7        Find the last definition X^p of v
8        for i := k to p + 1 do
9           add v to the set of influencing variables for action X^i
10       endfor
11    endfor
12 endfor
13 Display the annotated execution trace
```

Figure 4.7.3.1 Algorithm to Compute Influencing Variables

The algorithm begins by finding the last definition X^p of the slicing criterion. Once it has found it, variable y is added to the set of influencing variables for all actions between positions q and $p + 1$. The reason it stops at $p + 1$ and not at p is because the influencing variables at a particular position are the variables which influence the slicing criterion before the action is executed.

In the second part of the algorithm, the last definition X^p of every variable v used at each of the contributing action X^k is found. Like in the previous part, v is then added to the set of influencing variables for all actions between positions k and $p + 1$.

The comparison of the influencing variables for the baseline and modified versions of a program is also done using Beyond Compare. Figure 4.7.3.2 shows the influencing variables for the two versions of the `CoinBox` class.

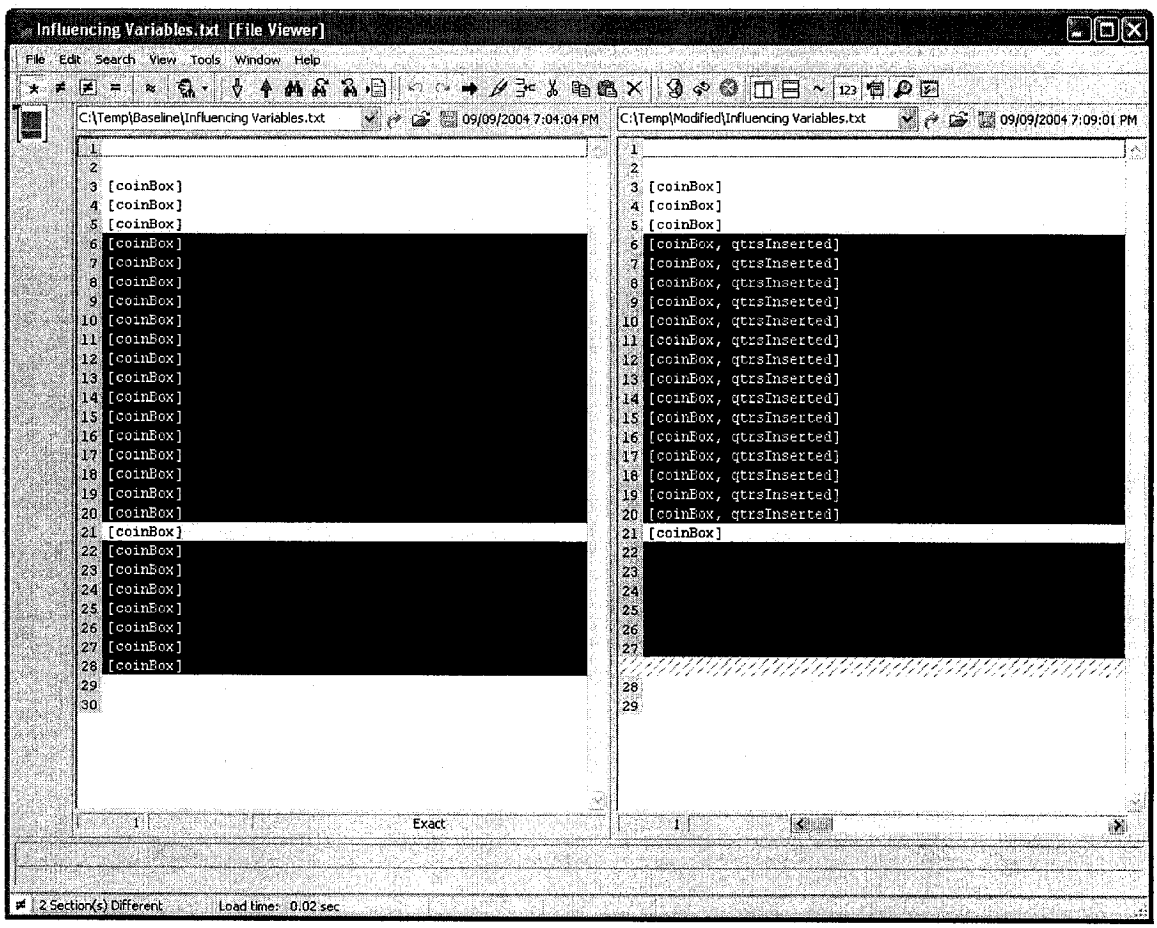

Figure 4.7.3.2 Comparison of Influencing Variables in Beyond Compare

The fine-grained slicing based testing approach also consists of comparing the contributing actions computed by the dynamic program slicing algorithm. However, this comparison has to be done separately, as illustrated in Figure 4.7.3.3.

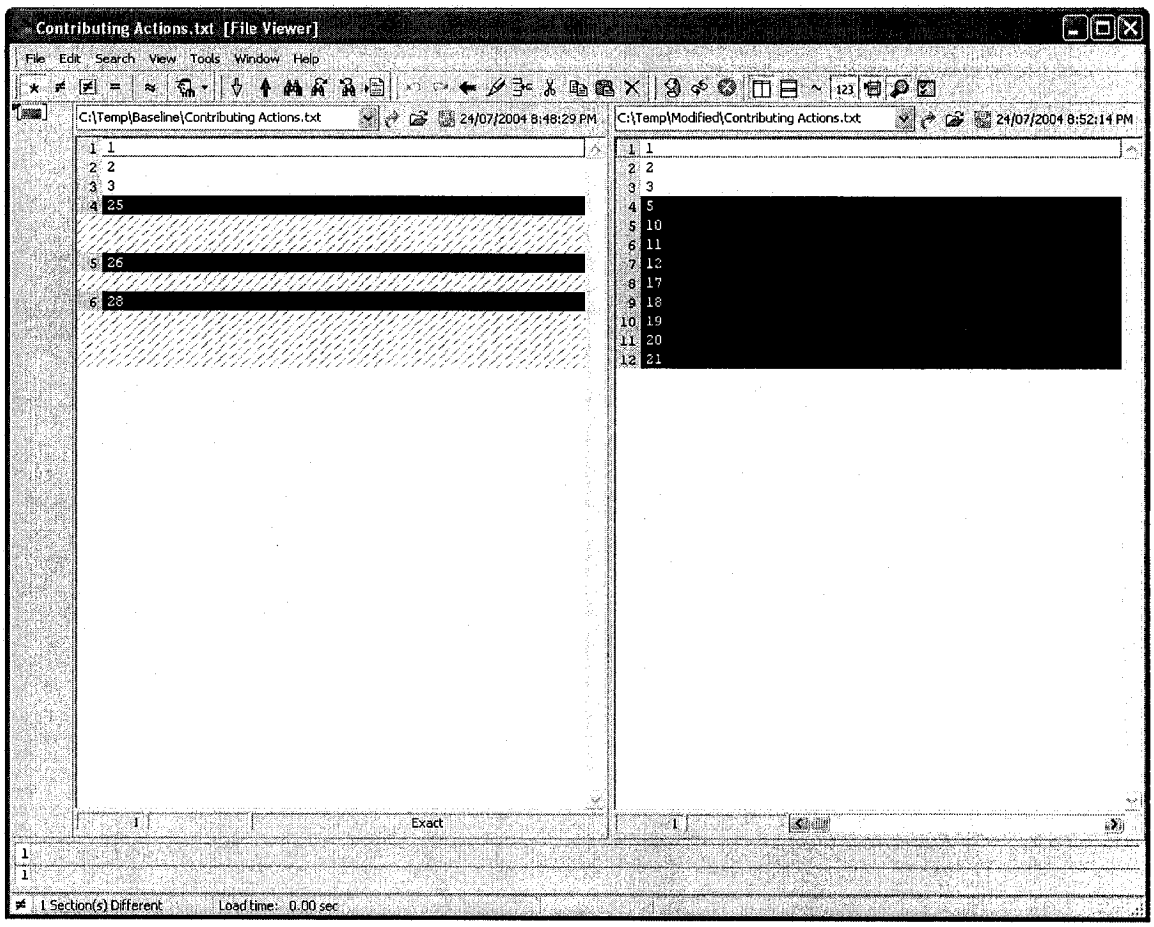

Figure 4.7.3.3 Comparison of Contributing Actions in Beyond Compare

4.8 LIMITATIONS

Like almost all research projects, the implementation of this one is not perfect and has some limitations. The most important ones are discussed next.

4.8.1 Execution Based Testing

As mentioned in section 4.5.3, an execution trace consists only of a file key and a line number for each executed source code line. Although the key is the same for both the

baseline and modified versions of a file, there may not be such a correlation in the case of the line numbers. For example, if a line of source code is removed or added in the modified version of a program, then all the line numbers coming after will be shifted up or down with respect to the baseline version. As a result, the comparison of the two execution traces will show several differences and will therefore complicate the analysis of the results. Though this limitation seems to be serious enough, execution traces are not recorded to be only compared. They are also required in order to compute dynamic slices, contributing actions, as well as influencing variables.

4.8.2 Inner Classes

Although the program slicing algorithm is dynamic, it still has to rely on static information generated by the parser. This is because the dynamic information provided by the instr_trace program is minimal: for each executed source code line, the output consists only of the file name and line number.

Presently, the parser cannot handle inner classes. As a result, the dynamic program slicing algorithm cannot compute slices for Java programs which make use of inner and anonymous inner classes.

4.8.3 Instance Variables

Another limitation of the parser which affects the program slicing algorithm involves objects. When a method is invoked on an object, the parser cannot determine whether or not new values are defined for its instance variables. The only information it provides is that the object is used.

The consequence of the above issue is as follows: if the last definition X^p of object y^q needs to be found, then all actions between positions p and q in which y is read have to be marked as contributing.

Execution Trace	Actions State
...	
45^{2} `CoinBox coinBox = new CoinBox();`	`contributing`
...	
46^{10} `coinBox.printTotal();`	`contributing`
...	
47^{17} `coinBox.insertQtr();`	`contributing`
...	

Figure 4.8.3.1 Execution Trace of a Sample Program with the Contributing Actions

For example, in Figure 4.8.3.1, if the last definition of object `coinBox` at position 17 has to be found, then the action at position 10 is also marked as contributing, even though the `printTotal()` method only reads the instance variable `qtrsInserted`.

This limitation of the parser results in the fact that the program slicing algorithm does not always compute the smallest slice. Sometimes, the slice contains statements which do not contribute to the computation of the variable of interest.

4.8.4 Short-Circuit Logical Operators

In Java, & and | are respectively the logical AND and OR operators. There exists also secondary versions of these and they are known as the short-circuit logical operators: && and | |. If the && and | | forms are used instead of the single-character versions, Java will not evaluate the right-hand operand when the outcome of the expression can be determined by the left operator alone. For example, in Figure 4.8.4.1, if `denom` is zero, there is no risk of causing a run-time exception, since the short-circuit form of AND is used.

```
if (denom != 0 && num / denom > 0)
```

Figure 4.8.4.1 Example of Short-Circuit Logical AND Operator

However, there is one problem with the use of the short-circuit versions of the logical operators. The control flow of the program cannot be determined statically when there is a method invoked by either the left or right operator. This in turn complicates the generation of the set of block traces, since it relies to a great extent on the static information provided by the parser. As a result, only the single-character version of both the AND and OR operators are supported by the program slicing algorithm.

4.8.5 Non-Executable Slices

Another limitation of the program slicing algorithm is that the slices it computes are not always executable. For example, in Figure 4.8.5.1, the interface `Callback` declares the `callback()` method, which is then implemented by the `Client` class. If, during the computation of a slice, the algorithm removes the implementation of the `callback()` method, then the resulting class will not compile.

```
interface Callback {
  void callback(int param);
}

class Client implements Callback {
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }
}
```

Figure 4.8.5.1 Interface Definition and Implementation

The same situation happens in Figure 4.8.5.2. The subclass *B* extends the abstract superclass *A*. If the implementation of the `callme()` method is removed during a slice computation, then the resulting subclass will not compile.

```
abstract class A {
  abstract void callme();
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
}
```

Figure 4.8.5.2 Abstract Method Definition and Implementation

4.8.6 Memory Requirements

As mentioned previously, in order to compute slices, the program slicing algorithm needs static and dynamic information about a Java program of interest. Currently, this information is stored in memory. As a result, the algorithm cannot compute slices for programs which have execution traces of several million statements, since this would require a prohibitive amount of memory.

4.9 Applicability

The three testing approaches proposed in this research help analyze the run time behavior of programs and as a result, detect possible faults. However, they are not substitutes for the traditional functional and structural testing techniques nor are they applicable to every situation. This is because there is a non-negligible overhead associated with each of them. This overhead consists of parsing the Java program of interest, collecting the run time information using the instrumentation toolkit, as well as computing the necessary dynamic slices, contributing actions, and influencing variables. Furthermore, once these have been computed, they have to be compared and analyzed in order to detect possible behavioral faults. This process can take a significant amount of time and cannot be fully automated.

Since all of the above activities consume time and effort, the proposed behavioral testing approaches should preferably be applied to the safety and mission critical components of

a system, as a complement to the already existing functional and structural testing techniques. For example, in the `CoinBox` class, they were successfully applied to the `vendingEnabled` instance variable, as it controls whether or not vending should be allowed. By limiting their use to the most important functions of a system, the benefits provided by these testing techniques should outweigh their associated costs.

5. CASE STUDY

To demonstrate the applicability of the different testing approaches presented in this research, a case study was performed using JUnit [BEC98]. This particular framework was selected because its source code is available and it has reached a rather mature development level.

5.1 JUNIT

JUnit is an open source Java testing framework originally written by Erich Gamma and Kent Beck. It is used to write and run repeatable test cases. It is an instance of the xUnit architecture for unit testing frameworks. JUnit consists of 100 class and interface definitions.

Unit testing consists of testing each component of a system independently to ensure that they all operate correctly [SOM01]. In object-oriented programming, a unit can be a class, several related classes, or an executable binary file [BIN00]. The purpose of unit testing is to uncover faults in the source code before the components are integrated into a complete system or subsystem, since the sooner faults are discovered, the easier it is to correct them.

JUnit defines how to structure unit test cases as well as provides the tools to run them, record their results, and report errors. Its features include: assertions for testing expected

results, test fixtures for sharing common test data, test suites for easily organizing and running tests, as well as graphical and textual test runners [JUN04].

5.2 DESCRIPTION OF THE FAULT

In JUnit, the class responsible for displaying the results generated by the text based test runner is `ResultPrinter`. A partial listing of its source code is shown in Figure 5.2.1.

```
 1 package junit.textui;
 2
 3 import java.io.PrintStream;
 4 import java.text.NumberFormat;
 5 import java.util.Enumeration;
 6
 7 import junit.framework.AssertionFailedError;
 8 import junit.framework.Test;
 9 import junit.framework.TestFailure;
10 import junit.framework.TestListener;
11 import junit.framework.TestResult;
12 import junit.runner.BaseTestRunner;
13
14 public class ResultPrinter implements TestListener {
15   PrintStream fWriter;
16   int fColumn;
17
18   public ResultPrinter(PrintStream writer) {
19     fColumn = 0;
20     fWriter = writer;
21   }
 ...
```

Figure 5.2.1 ResultPrinter Class

One of the instance variables of `ResultPrinter` is `fWriter`, which is of type `PrintStream`. It is through this instance variable that the `TestRunner` class of Figure 5.2.2 outputs the results of the test cases.

```
  1 package junit.textui;
  2
  3
  4 import java.io.PrintStream;
  5
  6 import junit.framework.*;
  7 import junit.runner.*;
  8
  …
 25 public class TestRunner extends BaseTestRunner {
 26   private ResultPrinter fPrinter;
 27
 28   public static final int SUCCESS_EXIT;
 29   public static final int FAILURE_EXIT;
 30   public static final int EXCEPTION_EXIT;
 31
 32   static {
 33     SUCCESS_EXIT   = 0;
 34     FAILURE_EXIT   = 1;
 35     EXCEPTION_EXIT = 2;
 36   }
 37
 38   /**
 39   * Constructs a TestRunner.
 40   */
 41   public TestRunner() {
 42     fPrinter = new ResultPrinter(System.out);
 43   }
  …
120   public static void main(String args[]) {
121     TestRunner aTestRunner = new TestRunner();
  …
```

Figure 5.2.2 TestRunner Class

The class `TestRunner` is a command line based tool used to run tests. One of its instance variables is `fPrinter` of type `ResultPrinter`. The `TestRunner` constructor initializes it by passing the standard output stream as an argument to the `ResultPrinter` constructor. As a result, the text based test runner outputs its results on the console, as illustrated in Figure 5.2.3.

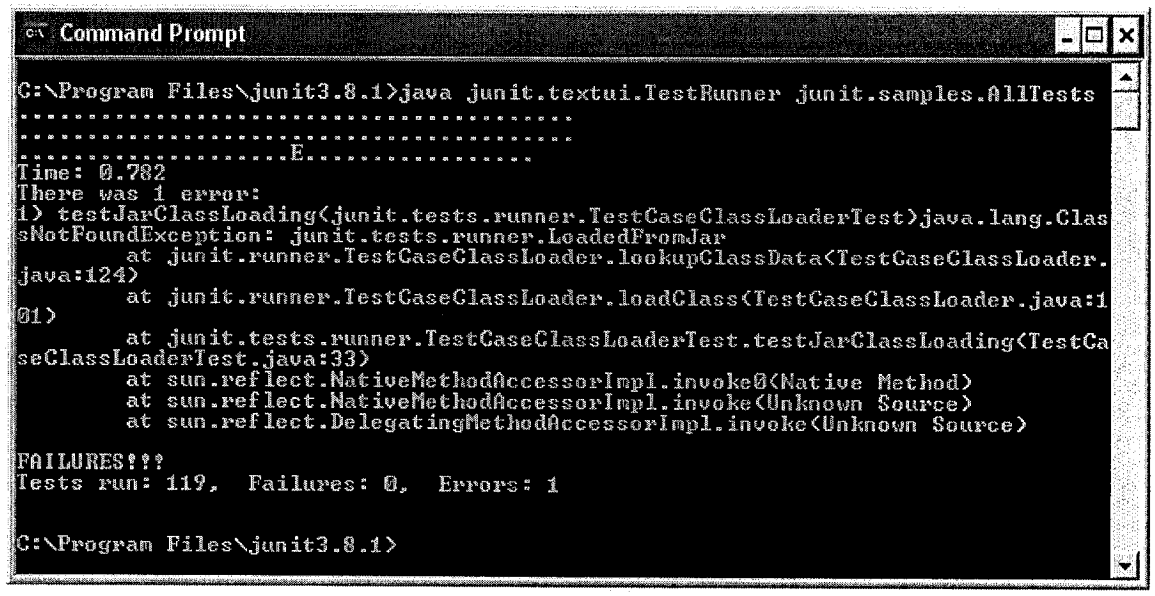

Figure 5.2.3 Test Results Generated by JUnit

In order to give the programmers the possibility to save the results of their test cases to a persistent storage area, JUnit was modified. The changes are highlighted in bold in Figure 5.2.4.

```
 1 package junit.textui;
 2
 3
 4 import java.io.PrintStream;
 5
 6 import junit.framework.*;
 7 import junit.runner.*;
 8
 …
25 public class TestRunner extends BaseTestRunner {
26   private ResultPrinter fPrinter;
27
28   public static final int SUCCESS_EXIT;
29   public static final int FAILURE_EXIT;
30   public static final int EXCEPTION_EXIT;
31
32   static {
33     SUCCESS_EXIT   = 0;
34     FAILURE_EXIT   = 1;
35     EXCEPTION_EXIT = 2;
36   }
37
38   /**
39   * Constructs a TestRunner.
40   */
```

Figure 5.2.4 TestRunner Class - Baseline Version

```
41   public TestRunner() {
42     String fileName = null;
43     System.out.print("Save Results As: ");
44     try {
45       fileName = new BufferedReader(new InputStreamReader(System.in)).readLine();
46       fPrinter = new ResultPrinter(new PrintStream(new BufferedOutputStream(new
         FileOutputStream(fileName)), true));
47     }
48
49     catch(FileNotFoundException e) {
50       System.err.println("Cannot open the file " + fileName);
51       System.exit(1);
52     }
53
54     catch(IOException e) {
55       System.err.println("Cannot read console input");
56       System.exit(1);
57     }
58   }
 …
135  public static void main(String args[]) {
136    TestRunner aTestRunner = new TestRunner();
 …
```

Figure 5.2.4 TestRunner Class - Baseline Version (Continued)

When the constructor of the `TestRunner` class is invoked, the user is prompted to enter the name of the file to which the results should be written. The file name is then passed as an argument to the `ResultPrinter` constructor.

In Figure 5.2.5, a fault was introduced in the `TestRunner` class. At line 46, the variable `fileName` was commented out and a string literal specifying the name of the file to open was hard-coded instead.

```
 1 package junit.textui;
 2
 3
 4 import java.io.PrintStream;
 5
 6 import junit.framework.*;
 7 import junit.runner.*;
 8
 …
25 public class TestRunner extends BaseTestRunner {
26   private ResultPrinter fPrinter;
27
28   public static final int SUCCESS_EXIT;
```

Figure 5.2.5 TestRunner Class - Modified Version

```
29    public static final int FAILURE_EXIT;
30    public static final int EXCEPTION_EXIT;
31
32    static {
33      SUCCESS_EXIT   = 0;
34      FAILURE_EXIT   = 1;
35      EXCEPTION_EXIT = 2;
36    }
37
38    /**
39    * Constructs a TestRunner.
40    */
41    public TestRunner() {
42      String fileName = null;
43      System.out.print("Save Results As: ");
44      try {
45        fileName = new BufferedReader(new InputStreamReader(System.in)).readLine();
46        fPrinter = new ResultPrinter(new PrintStream(new BufferedOutputStream(new
          FileOutputStream(/*fileName*/"F:\\Results.txt")), true));
47      }
48
49      catch(FileNotFoundException e) {
50        System.err.println("Cannot open the file " + fileName);
51        System.exit(1);
52      }
53
54      catch(IOException e) {
55        System.err.println("Cannot read console input");
56        System.exit(1);
57      }
58    }
 ...
135   public static void main(String args[]) {
136     TestRunner aTestRunner = new TestRunner();
 ...
```

Figure 5.2.5 TestRunner Class - Modified Version (Continued)

Even though this fault was introduced on purpose, it is believed that it is one which could easily occur in pratice: a software developer or tester hard-codes input data instead of always having to enter them each time the program is executed and forgets to remove them. If the program is always tested with input values which are the same as the hard-coded ones, then this fault could escape detection.

The three testing approaches proposed in the present research were applied successively to determine if they could help in uncovering the fault introduced in JUnit.

5.3 Execution Based Testing

The execution trace of the baseline and modified versions of JUnit were recorded and compared, as illustrated in Figure 5.3.1. In the present case, since no statement was added or removed, the execution based testing approach cannot uncover the fault because the two execution traces are identical.

Baseline Version	Modified Version
...	...
135[33] public static void main(String args...)	135[33] public static void main(String args...)
136[34] TestRunner aTestRunner = new TestRun...	136[34] TestRunner aTestRunner = new TestRun...
41[35] public TestRunner()	41[35] public TestRunner()
42[36] String fileName = null;	42[36] String fileName = null;
43[37] System.out.print("Save Results As:");	43[37] System.out.print("Save Results As:");
44[38] try {	44[38] try {
45[39] fileName = new BufferedReader(new ...	45[39] fileName = new BufferedReader(new ...
46[40] fPrinter = new ResultPrinter(new ...	46[40] fPrinter = new ResultPrinter(new ...
18[41] public ResultPrinter(PrintStream ...	18[41] public ResultPrinter(PrintStream ...
19[42] fColumn= 0;	19[42] fColumn= 0;
20[43] fWriter= writer;	20[43] fWriter= writer;
...	...

Figure 5.3.1 Execution Trace of JUnit's Baseline and Modified Versions

5.4 Coarse-Grained Slicing Based Testing

A dynamic slice was computed for both versions of JUnit with respect to the instance variable `fPrinter` of the `TestRunner` class at position 46. The comparison of the two resulting slices reveals the presence of a fault in the Modified version, since lines 42 and 45 are not part of its slice while they should have been.

Baseline Version	Modified Version
…	…
25 public class TestRunner extends …	25 public class TestRunner extends …
41 public TestRunner() {	41 public TestRunner() {
42 String fileName = null;	
44 try {	44 try {
45 fileName = new BufferedReader(new…	
46 fPrinter = new ResultPrinter(new …	46 fPrinter = new ResultPrinter(new …
47 }	47 }
135 public static void main(String args…	135 public static void main(String args…
136 TestRunner aTestRunner= new TestRun…	136 TestRunner aTestRunner= new TestRun…
146 }	146 }
187 }	… }

Figure 5.4.1 Slice of JUnit's Baseline and Modified Versions

5.5 Fine-Grained Slicing Based Testing

For the last testing approach, a dynamic slice was computed for variable `aTestRunner` at position 136 in the `TestRunner` class. The list of contributing actions and influencing variables were also identified at each point of the program's execution, as illustrated in Figures 5.5.1 and 5.5.2.

Contributing Actions	Influencing Variables
…	
135^{33} public static void main(String args[])	
136^{34} TestRunner aTestRunner = new TestRunner();	
41^{35} public TestRunner()	
42^{36} String fileName = null;	
43^{37} System.out.print("Save Results As: ");	
44^{38} try {	
45^{39} fileName = new BufferedReader(new InputStreamReader(System.in)) …	
46^{40} fPrinter = new … FileOutputStream(fileName)) …	fileName
18^{41} public ResultPrinter(PrintStream writer)	
19^{42} fColumn= 0;	writer
20^{43} fWriter= writer;	writer
…	

Figure 5.5.1 Contributing Actions and Influencing Variables for the Baseline Version

Contributing Actions	Influencing Variables
... 135[33] public static void main(String args[]) 136[34] TestRunner aTestRunner = new TestRunner(); 41[35] public TestRunner() 42[36] String fileName = null; 43[37] System.out.print("Save Results As: "); 44[38] try { 45[39] fileName = new BufferedReader(new InputStreamReader(System.in)) ... 46[40] fPrinter = new ... FileOutputStream(/*fileName*/"F:\\Results.txt")) ... 18[41] public ResultPrinter(PrintStream writer) 19[42] fColumn= 0; 20[43] fWriter= writer; ...	 writer writer

Figure 5.5.2 Contributing Actions and Influencing Variables for the Modified Version

Like the previous approach, this one uncovers the fault in the modified version of JUnit. It indicates that the variable `fileName` has no influence at position 46, even though `fPrinter` should use the value which was assigned to it previously, in order to open the file specified by the user.

5.6 Measurement of the Associated Overhead

The time needed to parse JUnit into the database, collect its run time information, as well as compute the slices, contributing actions, and influencing variables is indicated in the table below. These data where obtained using a 2.66 GHz Pentium 4 with 1024 MB of memory running Windows XP, PostgreSQL 7.3.6, and Java 2 SDK 1.4.2.

Testing Task	Time in Seconds
Parse JUnit into the database	66.30
Collect the run time information and store it into the database	0.33 + 23,83
Compute the slice for variable `fPrinter` in class `TestRunner`	10,312
Compute the contributing actions and influencing variables for variable `aTestRunner` in class `TestRunner`	10,281

Table 5.6.1 Overhead of the Testing Techniques

6. Conclusions and Future Work

After changes have been made to a software system, regression testing has to be performed. Its purpose is to validate the parts of the software which were modified, while at the same time ensuring that no new faults were introduced into previously tested code. In this research, three testing approaches were presented to provide additional confidence that the run time behavior of a program has not been inadvertently affected by a software modification. These approaches are execution based testing, coarse-grained slicing based testing, and fine-grained slicing based testing.

The three testing techniques described, as well as the underlying dynamic program slicing algorithm, were implemented as part of the CONCEPT research project using the Java programming language. They can analyze the run time behavior of object-oriented programs written in Java. The program slicing algorithm implemented is a modified version of the dynamic program slicing algorithm with removable blocks presented by Korel [KOR95, KOR97A].

A case study was also performed using the JUnit testing framework. It provided some insight into the types of faults that the proposed testing approaches can uncover and which might escape from the traditional functional and structural testing techniques.

However, in spite of their advantages, these techniques are not applicable to every situation because of their associated overhead. As a result, to be cost effective, their use

should be limited to the most important functions of a system and therefore, complement the traditional functional and structural testing techniques.

In addition to addressing the limitations discussed in Section 4.8, future work will focus on finding a metaphor to visualize the results generated by the different testing approaches. Currently, they are displayed in a text based format, but this is inadequate since it provides a large amount of information without any abstractions. The proposed testing techniques should also be applied to industrial production software to validate their performance, applicability, and usability.

7. REFERENCES

[AGR90] Agrawal, H. and J.R. Horgan, "Dynamic Program Slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, June 1990, pp. 246-256.

[AGR93A] Agrawal, H., R. DeMillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software - Practice and Experience*, vol. 23, no. 6, June 1993, pp. 589-616.

[AGR93B] Agrawal, H., et al., "Incremental Regression Testing," *Proceedings of the IEEE Conference on Software Maintenance*, Montréal, Canada, September 1993, pp. 348-357.

[AGR94] Agrawal, H., "On Slicing Programs with Jump Statements," *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994, pp. 302-312.

[BAT93] Bates, S. and S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993, pp. 384-396.

[BEC93] Beck, J. and D. Eichmann, "Program and Interface Slicing for Reverse Engineering," *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, May 1993, pp. 509-518.

[BEC98] Beck, K. and E. Gamma, "Test Infected: Programmers Love Writing Tests," *Java Report*, vol. 3, no. 7, July 1998, pp. 37-50.

[BEI90] Beizer, B., *Software Testing Techniques*, 2nd edition, Van Nostrand Reinhold, 1990.

[BIN94] Binder, R.V., "Object-Oriented Software Testing," *Communications of the ACM*, vol. 37, no. 9, September 1994, pp. 28-29.

[BIN00] Binder, R.V., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 2000.

[BINK97] Binkley, D.W., "Semantics Guided Regression Test Cost Reduction," *IEEE Transactions on Software Engineering*, vol. 23, no. 8, August 1997, pp. 498-516.

[BINK98] Binkley, D.W., "The Application of Program Slicing to Regression Testing," *Information and Software Technology*, vol. 40, no. 11-12, November 1998, pp. 583-594.

[CAN98] Canfora, G., A. Cimitile, and A. De Lucia, "Conditioned Program Slicing," *Information and Software Technology*, vol. 40, no. 11-12, November 1998, pp. 595-607.

[CHO91] Choi, J.-D., et al., "Techniques for Debugging Parallel Programs with Flowback Analysis," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, October 1991, pp. 491-530.

[CHU02] Chung, I.S., et al., "Abstract Program Slicing," *Proceedings of the 20th IASTED International Conference on Applied Informatics (AI '02)*, Innsbruck, Austria, February 2002, pp. 74-79.

[DEU82] Deutsch, M.S., *Software Verification and Validation*, Prentice Hall, 1982.

[DUE92] Duesterwald, E., R. Gupta, and M.L. Soffa, "Rigorous Data Flow Testing through Output Influences," *Proceedings of the 2nd Irvine Software Symposium (ISS '92)*, Irvine, California, March 1992, pp. 131-145.

[FER87] Ferrante, J., K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, July 1987, pp. 319-349.

[FRA88] Frankl, P.G. and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, October 1988, pp. 1483-1498.

[FRA93] Frankl, P.G. and E.J. Weyuker, "A Formal Analysis of the Fault-Detecting Ability of Testing Methods," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, March 1993, pp. 202-213.

[FRA97] Frankl, P.G., S.N. Weiss, and C. Hu, "All-Uses vs. Mutation Testing: An Experimental Comparison of Effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, September 1997, pp. 235-253.

[FRI92] Fritzson, P., "Generalized Algorithmic Debugging and Testing," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, December 1992, pp. 303-322.

[GAL91] Gallagher, K.B. and J.R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, August 1991, pp.751-761.

[GLE04] Glen McCluskey & Associates LLC, "Java Test Coverage and Instrumentation Toolkits," http://www.glenmccl.com/instr/, 2004.

[GOP91] Gopal, R., "Dynamic Program Slicing Based on Dependence Relations," *Proceedings of the Conference on Software Maintenance*, Sorrento, Italy, 1991, pp. 191-200.

[GUP92] Gupta R., M.J. Harrold, and M.L. Soffa, "An Approach to Regression Testing Using Slicing," *Proceedings of the Conference on Software Maintenance (CSM '92)*, Orlando, Florida, November 1992, pp. 299-308.

[HAR95] Harman, M. and S. Danicic, "Using Program Slicing to Simplify Testing," *Journal of Software Testing, Verification and Reliability*, vol. 5, no. 3, September 1995, pp. 143-162.

[HAR97] Harman, M. and S. Danicic, "Amorphous Program Slicing," *Proceedings of the 5th International Workshop on Program Comprehension (IWPC '97)*, Dearborn, Michigan, May 1997, pp. 70-79.

[HAR01A] Harman, M., et al., "Node Coarsening Calculi for Program Slicing," *8th IEEE Working Conference on Reverse Engineering (WCRE '01)*, Stuttgart, Germany, October 2001, pp. 25-34.

[HAR01B] Harman, M., et al., "Pre/Post Conditioned Slicing," *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '01)*, Florence, Italy, November 2001, pp. 138-147.

[HEU02] Heuzeroth, D., T. Holl, and W. Löwe, "Combining Static and Dynamic Analyses to Detect Interaction Patterns," *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT '02)*, Pasadena, California, June 2002.

[HIE99] Hierons, R.M., M. Harman, and S. Danicic, "Using Program Slicing to Assist in the Detection of Equivalent Mutants," *Journal of Software Testing, Verification, and Reliability*, vol. 9, no. 4, December 1999, pp. 233-262.

[HIE02] Hierons, R.M., et al., "Conditioned Slicing Supports Partition Testing," *Journal of Software Testing, Verification and Reliability*, vol. 12, no. 1, March 2002, pp. 23-28.

[HOF95] Hoffman, D. and P. Strooper, "The Testgraph Methodology: Automated Testing of Collection Classes," *Journal of Object-Oriented Programming*, vol. 8, no. 7, November-December 1995, pp. 35-41.

[HOR90] Horwitz, S., T. Reps, and D.W. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, January 1990, pp. 26-60.

[HOW80] Howden, W.E., "Functional Program Testing," *IEEE Transactions on Software Engineering*, vol. 6, no. 1, January 1980, pp. 162-169.

[IEE87A] *ANSI/IEEE Standard 610.12-1990: Glossary of Software Engineering Terminology*, The Institute of Electrical and Electronic Engineers, 1987.

[IEE87B] *ANSI/IEEE Standard 1008-1987: IEEE Standard for Software Unit Testing*, The Institute of Electrical and Electronic Engineers, 1987.

[JAC94A] Jackson, D. and E.J. Rollins, "Abstraction Mechanisms for Pictorial Slicing," *Proceedings of the IEEE Workshop on Program Comprehension*, Washington, D.C., November 1994, pp. 82-88.

[JAC94B] Jackson, D. and E.J. Rollins, "A New Model of Program Dependences for Reverse Engineering," *Proceedings of the 2nd ACM Symposium on Foundations of Software Engineering*, New Orleans, Louisiana, December 1994, pp. 2-10.

[JAD04] JAD, "JAD Home Page," http://kpdus.tripod.com/jad.html, 2004.

[JUN04] JUnit, "JUnit FAQ," http://junit.sourceforge.net/doc/faq/faq.htm, 2004.

[KAM92] Kamkar, M., N. Shahmehri, and P. Fritzson, "Interprocedural Dynamic Slicing and its Application to Generalized Algorithmic Debugging," *Proceedings of the International Conference on Programming Language Implementation and Logic Programming (PLILP '92)*, Leuven, Belgium, August 1992.

[KAM93A] Kamkar, M., P. Fritzson, and N. Shahmehri, "Interprocedural Dynamic Slicing Applied to Interprocedural Data How Testing," *Proceedings of the IEEE Conference on Software Maintenance*, Montréal, Canada, September 1993, pp. 386-395.

[KAM93B] Kamkar, M., P. Fritzson, and N. Shahmehri, "Three Approaches to Interprocedural Dynamic Slicing," *EUROMICRO Journal of Microprocessing and Microprogramming*, vol. 38, no. 1-5, September 1993, pp. 625-636.

[KAZ99] Kazman, R. and S.J. Carrière, "Playing Detective: Reconstructing Software Architecture from Available Evidence," *Journal of Automated Software Engineering*, vol. 6, no. 2, April 1999, pp. 107-138.

[KAZ03] Kazman, R., L. O'Brien, and C. Verhoef, "Architecture Reconstruction Guidelines, Third Edition," *Technical Report CMU/SEI-2002-TR-034I*, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 2003.

[KOR88] Korel, B. and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, vol. 29, no. 3, October 1988, pp. 155-163.

[KOR94] Korel, B. and S. Yalamanchili, "Forward Derivation of Dynamic Slices," *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA '94)*, Seattle, Washington, August 1994, pp. 66-79.

[KOR95] Korel, B., "Computation of Dynamic Slices for Programs with Arbitrary Control-Flow," *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG '95)*, Saint-Malo, France, May 1995, pp. 71-86.

[KOR97A] Korel, B., "Computation of Dynamic Program Slices for Unstructured Programs," *IEEE Transactions on Software Engineering*, vol. 23, no. 1, January 1997, pp. 17-34.

[KOR97B] Korel, B. and J. Rilling, "Dynamic Program Slicing in Understanding of Program Execution," *Proceedings of the 5th International Workshop on Program Comprehension (IWPC '97)*, Dearborn, Michigan, May 1997, pp. 80-89.

[KRI94] Krishnaswamy A., "Program Slicing: An Application of Object-Oriented Program Dependency Graphs," *Technical Report TR94-108*, Clemson University, Clemson, South Carolina, July 1994.

[KUN96] Kung, D., et al., "Object State Testing and Fault Analysis for Reliable Software Systems," *Proceedings of the 7th International Symposium on Software Reliability Engineering*, White Plains, New York, October 1996, pp. 76-85.

[KUN98] Kung, D.C., P. Hsia, and J. Gao, *Testing Object-Oriented Software*, IEEE Computer Society Press, 1998.

[LAR94] Larus, J.R. and S. Chandra, "Using Tracing and Dynamic Slicing to Tune Compilers," *Technical Report UW-CS 1174*, University of Wisconsin-Madison, Madison, Wisconsin, August 1994.

[LAR96] Larsen, L.D. and M.J. Harrold, "Slicing Object-Oriented Software," *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, March 1996, pp. 495-505.

[LAW94] Law, C.H.R., "Object-Oriented Program Slicing," *Ph.D. Thesis*, University of Regina, Regina, Canada, 1994.

[LEJ92] Lejter, M., S. Meyers, and S.P. Reiss, "Support for Maintaining Object-Oriented Programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, December 1992, pp. 1045-1052.

[LET86] Letovski, S. and E. Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software*, vol. 3, no. 3, May 1986, pp. 41-49.

[LOW01] Löwe, W., A. Ludwig, and A. Schwind, "Understanding Software - Static and Dynamic Aspects," *17th International Conference on Advanced Science and Technology (ICAST '01)*, Chicago, Illinois, October 2001, pp. 83-88.

[LYL87] Lyle, J.R. and M.D. Weiser, "Automatic Program Bug Location by Program Slicing," *Proceedings of the 2nd International Conference on Computers and Applications*, Peking, China, June, 1987, pp. 877-883.

[MAR94] Marick, B., *The Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing*, Prentice Hall, 1994.

[MOR92] Morell, L.J. and L.E. Deimel, "Unit Analysis and Testing," *Technical Report SEI-CM-9-2.0*, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1992.

[MYE79] Myers, G.J., *The Art of Software Testing*, John Wiley & Sons, 1979.

[OTT84] Ottenstein, K.J. and L.M. Ottenstein, "The Program Dependence Graph in a Software Development Environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, April 1984, pp. 177-184.

[PAN93] Pan, H., "Software Debugging with Dynamic Instrumentation and Test-Based Knowledge," *Ph.D. Thesis*, Purdue University, West Lafayette, Indiana, 1993.

[PER90] Perry, D.E. and G.E. Kaiser, "Adequate Testing and Object-Oriented Programming," *Journal of Object-Oriented Programming*, vol. 2, no. 5, January 1990, pp. 13-19.

[PRE01] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, 5th edition, McGraw-Hill, 2001.

[REP88] Reps, T. and S. Horwitz, "Semantics-Based Program Integration," *Proceedings of the 2nd European Symposium on Programming (ESOP '88)*, Nancy, France, March 1988, pp. 133-145.

[REP89] Reps, T. and T. Bricker, "Semantics-Based Program Integration Illustrating Interference in Interfering Versions of Programs," *Proceedings of the 2nd International Workshop on Software Configuration Management*, Princeton, New Jersey, October 1989, pp. 46-55.

[RIC94] Richardson, D.J., "TAOS: Testing with Analysis and Oracle Support," *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA '94)*, Seattle, Washington, August 1994, pp. 138-153.

[RIL98] Rilling, J., "Investigation of Program Slicing and its Applications in Program Comprehension of Large Software Systems," *Ph.D. Thesis*, Illinois Institute of Technology, Chicago, Illinois, 1998.

[RIL01A] Rilling, J., "Maximizing Functional Cohesion of Comprehension Environments by Integrating User and Task Knowledge," *8th IEEE Working Conference on Reverse Engineering (WCRE '01)*, Stuttgart, Germany, October 2001, pp. 157-165.

[RIL01B] Rilling, J. and A. Seffah, "MOOSE - A Task-Driven Program Comprehension Environment," *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC '01)*, Chicago, Illinois, October 2001, pp.77-86.

[RIL02] Rilling, J., A. Seffah, and C. Bouthlier, "The CONCEPT Project - Applying Source Code Analysis to Reduce Information Complexity of Static and Dynamic Visualization Techniques," *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, Paris, France, June 2002, pp. 90-99.

[ROT94] Rothermel, G. and M.J. Harrold, "Selecting Tests and Identifying Test Coverage Requirements for Modified Software," *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA '94)*, Seattle, Washington, August 1994, pp. 169-184.

[SCO04] Scooter Software, "Beyond Compare," http://www.scootersoftware.com/, 2004.

[SMI90] Smith, M.D. and D.J. Robson, "Object-Oriented Programming: The Problems of Validation," *Proceedings of the IEEE Conference on Software Maintenance*, San Diego, California, November 1990, pp. 272-281.

[SOM01] Sommerville, I., *Software Engineering*, 6th edition, Addison-Wesley, 2001.

[SUN04] Sun Microsystems, "Java Platform Debugger Architecture," http://java.sun.com/products/jpda/, 2004.

[TIP95] Tip, F., "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, vol. 3, no. 3, September 1995, pp.121-189.

[VAN00] Van Vliet, H., *Software Engineering: Principles and Practice*, 2nd edition, John Wiley & Sons, 2000.

[WEI82] Weiser, M.D., "Programmers Use Slices when Debugging," *Communications of the ACM*, vol. 25, no. 7, July 1982, pp. 446-452.

[WEI83] Weiser, M.D., "Reconstructing Sequential Behaviour from Parallel Behaviour Projections," *Information Processing Letters*, vol. 17, no. 10, 1983, pp. 129-135.

[WEI84] Weiser, M.D., "Program Slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, July 1984, pp. 352-357.

[WIL92] Wilde, N. and R. Huitt, "Maintenance Support for Object-Oriented Programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, December 1992, pp. 1038-1044.

[ZHA98] Zhao, J., "Dynamic Slicing of Object-Oriented Programs," *Technical Report SE-98-119*, Information Processing Society of Japan (IPSJ), Tokyo, Japan, May 1998, pp. 11-23.

[ZHA03] Zhang, Y., "Automatic Design Pattern Recovery," *Master's Thesis*, Concordia University, Montréal, Canada, 2003.