Bytecode-Based Analysis for Increasing Class-Component Testability

Supaporn Kansomkeat[1], **Jeff Offutt**[2], and **Wanchai Rivepiboon**[1], Non-members

ABSTRACT

Software testing attempts to reveal software faults by executing the program on input values and comparing the outputs from the execution with expected outputs. Testing software is easier when testability is high, so increasing testability allows faults to be detected more efficiently. Component-based software is often constructed from third party software components. When this is done, the reused components must be retested in the new environment to ensure that they integrate correctly into the new context. However, the "black box" nature of reused components, although offering great benefits otherwise, implicitly reduces their testability. Therefore, it helps to increase a component's testability before it is reused. To increase a component's testability, we need information that can be gained through program analysis. A crucial property of reused software components is that the source is not available, making program analysis significantly more difficult. This research addresses this problem by performing program analysis at the bytecode level.

This **bytecode analysis technique** increases component testability without requiring access to the source. A component's bytecode is analyzed to gather control and data flow information, which is then used to obtain definition and use information of method and class variables. Then, the definition and use information is used to increase component testability during component integration testing. We have implemented the technique and present empirical results for some components, demonstrating that the method is feasible and practical.

Keywords: Software Testing, Software Testability, Component Software, Bytecode, Program Analysis

1. INTRODUCTION

Program analysis is a way to inspect programs to gather some properties such as control and data flow information. An early use was to support code optimization in compilers [1].

Program analysis has also been widely used for software engineering problems such as program understanding, testing, and maintenance. Program analysis is used in testing to precisely compute what parts need to be executed [25], to determine which test cases must be rerun to test the program after modifying [15], and to generate more effective tests [2].

Software testing is used to verify software quality and reliability, but it can be an expensive and labor-intensive task [6]. Software testing attempts to reveal software faults by executing the program on inputs and comparing the outputs of the execution with expected outputs. Many research papers have focused on methods to reduce the test effort [12, 15, 16]. An aspect of software that influences the test effort and success is known as *testability.*

Several different definitions of testability have been published. According to the 1990 IEEE standard glossary [17], testability is the "degree to which a component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met." Voas and Miller [26] defined software testability by focusing on the "probability that a piece of software will fail on its next execution during testing if the software includes a fault." Binder [7] defined testability in term of controllability and observability. *Controllability* is the ability that users have to control a component's inputs (and internal state). *Observability* is the ability that users have to observe a component's outputs. If users cannot **control** the inputs, they cannot be sure what caused a given output. If users cannot **observe** the output of a component under test, they cannot be sure if the execution was correct. Freedman [11] also described testability based on the notions of observability and controllability. In his terms, observability captures the degree to which a component can be observed to generate the correct output for a given input, and controllability refers to the ease of producing all values of its specified output domain.

Object-oriented software is increasingly used, partly because it emphasizes portability and reusability. Java classes are compiled into portable binary class files, *class-components*, which contain statements called *bytecode.* The class-components are included in Java libraries without source code, thus the source is not always available.

Manuscript received on January 5, 2006; revised on March 15, 2006.

[1]The authors are with the Software Engineering Laboratory, Department of Computer Engineering,Chulalongkorn University, Bangkok, Thailand 10330; E-mail: supaporn.k@student.chula.ac.th, wanchai.r@chula.ac.th

[2]The author is with the Information and Software Engineering Department,George Mason University, Fairfax, VA 22030, USA; E-mail: offutt@ise.gmu.edu

An important goal of reusable components is that the "re-users" should not need to understand how the components work, and should not need or want access to the source. Furthermore, it is usually assumed that the initial developers tested the component. However, this initial testing was either carried on the component in isolation (unit testing), in its original context, or both.

The goal of this research is to test a reused component with regards to how it integrates into a new context. That is, this is a form of integration testing that asks whether the component behaves appropriately in this new context. Component-based development organizations that we work with find behavioral mismatches to be a major source of failure in integration testing, system testing, and deployment.

Thus, when a component is reused, a key issue is how to test the component in its new context. Weyuker suggests that a component should be tested many times individually, and also each time it is integrated into a new system [28]. Voas and Miller [26] explained that testability enhances testing and claimed that increasing testability of components is crucial to improving the testability of component-based software. Wang et al. [27] increase component testability by using the built-in test (BIT) approach, that is, putting complete test cases inside the components. The tests are constantly presented and reused with the component. The disadvantage of the BIT approach is growth of programming overhead and component complexity. Naturally, component developers do not always provide BITs and testing information to component users. Instead of increasing testability by BIT, this research tests the component when it is integrated into a new context. Because of the lack of source code, program analysis techniques cannot be applied to the source. To address this problem, we apply program analysis at the *bytecode* level.

This paper presents an analysis technique at the bytecode level that is used to directly increase class-component testability without requiring access to the source. First, Java bytecode (.class file) is analyzed to extract control flows and data flows. This flow information is used to collect definition and use information of the component's method and class variables. Finally, the collected information is used to increase class-component testability.

The increased testability supports class-component testing by supporting the generation of tests to exercise a class-component in various ways (increasing controllability), thus faults can more easily be revealed. The increased testability also helps monitor the results of testing (increasing observability), thus class-component failures can more easily be detected. A previous paper [19] presented the analysis technique from bytecode intermediate form created by Decompiler [4]. This paper presents the bytecode-based analysis technique that directly analyzes Java bytecode. Also, this paper addresses the generation of test data and describes how our ideas can be used to increase observability.

The remainder of this paper is organized as follows. Section 2 presents background concepts. Section 3 describes the process of increasing class-component testability. Then, a case study is shown in section 4 and conclusions are presented in section 5.

2. BACKGROUND

This paper presents an analysis technique to increase class-component testability. The analysis is carried out at the bytecode level. The bytecode instructions are parsed to collect information based on data flow analysis. This information provides ways to generate test inputs and observe the outputs of testing. Coupling-based criteria are used to guide test selection and to monitor the results of tests. Thus, this section provides brief overviews of these topics.

2.1 Java Bytecode Instruction

Java programs are written and compiled into portable binary class files. Each class is represented by a single file that contains class related data and bytecode instructions. A Java runtime system dynamically loads the bytecode into an interpreter (Java Virtual Machine, JVM) and executes it. An example Java class and its corresponding bytecode instructions of method *calCoeff* are shown in Fig. 1.

The Java bytecode instructions can be roughly grouped as follows:

• *Stack operations*: Pushing constants onto the stack (e.g. *ldc*, *iconst_5*, *bipush*).

• *Arithmetic operations*: There are different arithmetic instructions for the different types in Java (e.g. *iadd*, *fmul*, *lneg*).

• *Control flow*: There are unconditional branch instructions (e.g. *goto*, *isr*, *ret*) and conditional branch instructions (e.g. *ifne*, *if_icmpeq*, *ifnull*).

• *Load and store operations*: The load and store instructions transfer values between the local variables and the operand stack (e.g. *istore*, *astore_1, aload_0*).

• *Field access*: Accessing fields of classes and fields of class instances (*getfield*, *putfield*, *getstatic*, *putstatic*).

• *Method invocation and return*: Calling methods (e.g. *invokestatic*, *invokevirtual*, *invokespecial*) and returning methods (e.g. *return*, *ireturn*, *areturn*).

• *Object allocation*: Allocating objects (e.g. *new*, *newarray*, *multianewarray*).

• *Conversion and type checking*: Checking and converting basic types and instances (e.g. *i2f*, *checkcast*, *instanceof*).

• *Operand Stack Management*: Directly manipulating the operand stack (e.g. *pop*, *swap*, *dup*).

• *Throwing Exception*: Exceptions are thrown using the *athrow* instruction.

A list of all instructions with detailed description can be found in the JVM Specification [21].

```
1:  public class SetCoeff {
2:
3:    int co,rate;
4:
5:    void calCoeff() {
6:       try {
7:          if (rate == 0 )
8:             throw new  ArithmeticException();
9:          else   if ( rate < 20 )
10:                  co = 5;
11:         else
12:                  co = 15;
13:      } // try
14:      catch ( ArithmeticException e ) {
15:         System.out.println("Exception");
16:      } // catch
17:   } // calCoeff
18:  } // class
```

```
Method void calCoeff()

0:   aload_0          // Load var#0 (rate) onto stack
1:   getfield         Coeff.rate I (2)
4:   ifne   #15       // branch to address 15
7:   new              <java.lang.ArithmeticException> (3)
10:  dup
11:  invokespecial
               java.lang.ArithmeticException.<init>()V (4)
14:  athrow           // Throwing exception
15:  aload_0
16:  getfield         Coeff.rate I (2)
19:  bipush   20
21:  if_icmpge #32  // branch to address 32
24:  aload_0
25:  iconst_5
26:  putfield         Coeff.co I (5)
29:  goto   #38       // branch to address 38
32:  aload_0
33:  bipush  15
35:  putfield         Coeff.co I (5)
38:  goto   #50       // branch to address 50
41:  astore_1         // Exception handle block
42:  getstatic        java.lang.System.out
                      Ljava/io/PrintStream; (6)
45:  ldc              "Exception" (7)
47:  invokevirtual   java.io.PrintStream.println
                           (Ljava/lang/String;)V (8)
50:  return
```

Fig.1: *A Simple Java Calss and the Bytecode Instructions for Method CalCoeff*

In Java, an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. When an exception is raised, control transfers to a block of instructions that can handle the exception. This block of instructions is called an *exception handler*. For example, the catch block in Fig. 1 is an exception handler.

2.2 Data Flow Analysis

Data flow testing [20, 24] tries to ensure that the correct values are stored into memory and that they are subsequently used correctly. A definition (*def*) is a statement where a variable's value is stored into memory. A *use* is a statement where a variable's value is accessed. A *definition-use pair* (or dupair) of a variable is an ordered pair of a definition and a use, such that there is an execution path from the def to the use without any intervening redefinitions of the variable. Data flow criteria require tests to execute paths from specific definitions to uses by selecting particular definition-use pairs to test. Laski and Korel defined the first two data flow testing criteria [20]. They proposed the *all-definitions* criterion, which requires tests to cover a path from each definition to at least one use, and the *all-uses* criterion, which requires tests to cover a path from each def to **all reachable** uses.

2.3 Coupling-based Testing

At least two methods have been developed to apply data flow testing inter-procedurally. First, Harrold and Rothermel [14] proposed a complete control and data flow analysis to compute the *program dependency graph* (which combines control and data flows) for pairs of methods. Directly applying either the all-defs or the all-uses criterion in this way is expensive, both in terms of the number of du-pairs and the difficulty of resolving the paths. Jin and Offutt [18] proposed a simpler model that just focuses on the connections. Their *coupling-based testing* (*CBT*) applies data flow testing to the integration level by requiring the program to execute data transfers from definitions of variables in a caller to uses of the corresponding variables in the callee unit. Instead of all variables definitions and uses, *CBT* is only concerned with definitions of variables that are transmitted just **before** calls (*last-defs*) and uses of variables just **after** calls (*first-uses*). The criteria are based on the following definitions:

- A *Coupling-def* is a statement that contains a last-def that can reach a first-use in another method on at least one execution path
- A *Coupling-use* is a statement that contains a first-use that can be reached by a last-def in another method on at least one execution path
- A *coupling path* is a path from a coupling-def to a coupling-use

Jin and Offutt [18] defined four coupling-based integration test coverage criteria. This paper uses the *all-coupling-uses* criterion. All-coupling-uses requires, for **each coupling-def** of a variable in the caller, the test cases to cover at least one coupling path to **each reachable coupling-use**.

3. CLASS-COMPONENT TESTABILITY

This section gives details about the proposed method. First, a class-component is analyzed at the bytecode level to collect information based on data flow analysis, *bytecode-based class-component analysis.* Then, the collected information is used to increase class-component testability, *increasing class-component testability.* The details of each process will be described in the following subsections.

3.1 Bytecode-based Class-Component Analysis

Conventional program analysis collects control and data flow information from source code. Because of source code is not available, our analysis technique collects control and data flow information at the *bytecode* level.

The most common way to represent the control flow relation is a *Control Flow Graph* (*CFG*), originally proposed for compiler optimization [1]. Nodes in a *CFG* represent a statement or a basic block of statements, and edges represent the flow of control. A *Data Flow Graph* represents flows of data [24]; nodes represent statements as in a *CFG*, but an edge is drawn from one node to another if a variable definition at the origin node can reach a use at the target node. The *Program Dependency Graph* (*PDG*) [14] uses nodes that represent statements and combines edges from both control and data flow graphs.

```
  0:   aload_0          // BasicBlock0
  1:     getfield       Coeff.rate I (2)  //use rate
  4:   ifne             #15        // branch to address 15
-------------------------------------------------------------------
                        // BasicBlock1
  7:   new            <java.lang.ArithmeticException>
 10:  dup
 11:  invokespecial
                    java.lang.ArithmeticException.<init>
 14:  athrow            // Throwing exception
-------------------------------------------------------------------
 15:  aload_0           // BasicBlock2
 16:    getfield        Coeff.rate I (2)  // use rate
 19:  bipush            20
 21:  if_icmpge         #32        //branch to address 32
-------------------------------------------------------------------
 24:  aload_0           // BasicBlock3
 25:  iconst_5
 26:    putfield        Coeff.co I (5)  // define co
 29:  goto              #38        //branch to address 38
-------------------------------------------------------------------
 32:  aload_0           // BasicBlock4
 33:  bipush            15
 35:    putfield        Coeff.co I (5)  // define co
-------------------------------------------------------------------
                        // BasicBlock5
 38:  goto              #50 // branch to address 50
-------------------------------------------------------------------
                        // BasicBlock6
 41:  astore_1          // Exception handle block
 42:  getstatic         java.lang.System.out
                          Ljava/io/PrintStream; (6)
 45:  ldc               "Catch Exception" (7)
 47:  invokevirtual       java.io.PrintStream.println
                            (Ljava/lang/String;)V (8)
-------------------------------------------------------------------
 50:  return             // BasicBlock7
```

Fig.2: *Partitioning Method calCoeff in Fig. 1 into Basic Blocks*

This bytecode analysis in this research results in a *Control-Data Flow Graph* (*CDFG*) for each method of a class-component. The *CDFG* includes both control and data flow edges, like the *PDG,* but differs in representing basic blocks in nodes and being created quite differently. The *CDFG* also includes exception handling control, which was not defined for the *PDG*. The *CDFG* construction process is as follows. First, the instructions are extracted and partitioned into *basic blocks,* then the flows of control and data are added.

3.1.1 Determining Basic Blocks

A basic block is a sequence of consecutive program statements in which flow of control enters at the beginning and leaves at the end without halting or branching except at the end. If any instruction is executed in a basic block, all instructions will be executed. The bytecode represents individual programming statements as several bytecode instructions, so the original statements are somewhat obscured and the analysis has to determine statements as well as basic blocks. To create a basic block, the analysis identifies *leader instructions*, which are instructions that begin basic blocks:

- The first instruction is a **leader**
- Any instruction that is the target of a conditional or unconditional branch instruction (*ifeq*, *if_icmpne*, *ifnull*, *tableswitch*, *goto*, etc.) is a **leader**
- Any instruction that immediately follows a conditional or unconditional branch instruction is a **leader**
- Any instruction that immediately follows a return instruction (*return*, *ireturn*, *areturn*, etc.) or *athrow* (exception handling) is a **leader**
- The first instruction of an *exception handler* is a **leader**

After identifying a leader instruction, a basic block is defined as consisting of a leader and all instructions up to but not including the next leader. The *EXIT* block is added to be the exit point. For example, the bytecode instructions for method *calCoeff* in Fig. 1 are divided into the following basic blocks: [0-4], [7-14], [15-21], [24-29], [32-35], [38], [41-47], and [50] as shown in Fig. 2. Each basic block is analyzed to gather *getfield* and *putfield* instructions for data flow information. Each *getfield* instruction indicates a *use* and each *putfield* instruction indicates a *definition.*

3.1.2 Constructing the CDFGs

After each basic block has been defined, edges associated with the flow of control are added. An edge is added from basic block B1 to B2 according to the following rules, which are based on the last instruction in B1:

- An unconditional branch instruction: An edge is added from B1 to the basic block whose leader is the target of the branch instruction of B1 (e.g. from *BasicBlock3* to *BasicBlock5* in Fig. 2).

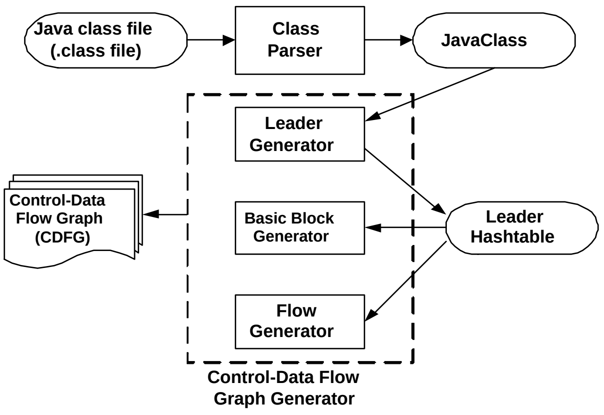
```
class VendingMachine {

  int total    = 0;
  int curQtr = 0;
  int Type    = 0;
  int[] availType =  new int[] {2,3,13};

  void addQtr() {
    curQtr = curQtr + 1;
  }

  void returnQtr() {
    curQtr = 0;
  }

 void vend ( int selection ) {
   int MAXSEL = 20;
   int VAL        = 2;
   Type           = selection;
   if ( curQtr == 0 )
       System.err.println ("No coins inserted");
   else if ( Type > MAXSEL )
       System.err.println ("Wrong selection ");
   else if ( !available() )
       System.err.println ("Selection  unavailable");
   else {
       if ( curQtr < VAL )
          System.err.println ("Not enough coins");
       else {
          System.err.println ("Take selection");
          total     = total+ VAL;
          curQtr = curQtr - VAL;
       }
   }
   System.out.println ("Current value=" + curQtr );
 }

 boolean available( ) {
   for (int i = 0; i<availType.length; i++)
      if (availType[i] == Type)
          return true;
   return false;
 }
} // class VendingMachine
```

Fig.3: *The Vending Machine Class*

• A conditional branch instruction: Two edges are added from B1. The first is to the basic block whose leader is the first instruction that directly follows the last instruction of B1 (e.g. from *BasicBlock0* to *BasicBlock1*). The second is to the basic block whose leader is the target of the branch instruction of B1 (e.g. from *BasicBlock0* to *BasicBlock2*).
• A return instruction: An edge is added from B1 to the exit point, *EXIT* (e.g. from *BasicBlock7* to the *EXIT* block).
• An *athrow* instruction: An edge is added from B1 to the basic block whose leader is the first instruction of the associated exception handler. If there is no associated exception handler, an edge is added to the exit, *EXIT* (e.g. from *BasicBlock1* to *BasicBlock6*).
• Not a branch, return or *athrow* instruction: This happens when a B1 ends just before a leader of another basic block. Add an edge from B1 to the next basic block (e.g. from *BasicBlock4* to *BasicBlock5*).

We illustrate our technique with a vending machine example taken from Harrold et al. [13]. The Java source code and bytecode instructions for the vending machine are shown in Fig. 3 and 4. The numbers of Fig. 4 indicate the instruction positions of the bytecode. Table 1 shows the *CDFG*s of vending machine in tabular form. Columns First-Inst and Last-Inst show the first and last instruction positions of each basic block. Column DefUse shows the sequence of definitions and uses of variables. Each element in the DefUse column contains *d* (a definition) or *u* (a use), the variable name, and the position of the instruction. For example, the element (*d, Type, 16*) in basic black 0 of method <*init*> indicates that the variable *Type* is defined (*putfield* instruction) at position 16. Column Pre-Block shows the basic block numbers that have the flows of control to the current block (predecessor block). Column Suc-Block shows the basic block numbers that have the incoming flows of control from the current block (successor block).

Data flow analysis is a way to obtain variables' relationships from flow graphs. This technique examines definitions and the subsequent uses of variables. Suppose instruction I_1 defines a value to x, which instruction I_2 then uses. Then, instructions I_1 and I_2 have a data flow relationship. From the previous step, data flow information can be collected by traversing the basic blocks in the *CDFG*s. For the vending machine example, the data flow relationship of variable *Type* within method *vend* is in basic block 0 at position 7 (def) and basic block 2 at position 29 (use). Data flow relationships can also be obtained between methods, for example, variable *curQtr* is defined in basic block 0 of method <*init*> and then used in basic block 0 of method *addQtr*.



Java class file (.class file)
Class Parser
JavaClass
Leader Generator
Control-Data Flow Graph (CDFG)
Basic Block Generator
Leader Hashtable
Flow Generator
Control-Data Flow Graph Generator

Fig.5: *The Class-component Analysis Process*

3.1.3 Appling the CDFGs

Our analysis process uses an open source tool from Apache/Jakarta [8] a bytecode manipulation library called BCEL (Byte Code Engineering Library). The BCEL API can analyze, create and manipulate Java bytecode files. Fig. 5 illustrates the class-component analysis process. There are two major components: (1) the *Class Parser* and (2) the *Control-Data Flow Graph Generator* (*CDFGen*). The *Class Parser* parses the Java class file and creates the *JavaClass* object, which represents all the information about

```
void <init>()
0:   aload_0
...
6:   putfield       VendingMachine.total I (2)
...
11:  putfield       VendingMachine.curQtr I (3)
...
16:  putfield       VendingMachine.Type I (4)
...
36:  putfield       VendingMachine.availType I (5)
39:  return

void addQtr()
...
2:   getfield       VendingMachine.curQtr I (3)
...
7:   putfield       VendingMachine.curQtr I (3)
10:  return

void returnQtr()
...
2:   putfield       VendingMachine.curQtr I (3)
5:   return

void vend (int arg1)
0:   bipush         20
...
7:   putfield       VendingMachine.Type I (4)
10:  aload_0
11:  getfield       VendingMachine.curQtr I (3)
14:  ifne           #28
...
25:  goto           #112
28:  aload_0
29:  getfield       VendingMachine.Type I (4)
32:  iload_2
33:  if_icmple      #47
...
44:  goto           #112
...

51:  ifne           #65
...
62:  goto           #112
65:  aload_0
66:  getfield       VendingMachine.curQtr I (3)
69:  iload_3
70:  if_icmpge      #84
...
81:  goto           #112
...
94:  getfield       VendingMachine.total I (2)
...
99:  putfield       VendingMachine.total I (2)
...
104: getfield       VendingMachine.curQtr I (3)
...
109: putfield       VendingMachine.curQtr I (3)
...
128: getfield       VendingMachine.curQtr I (3)
...
140: return

boolean available()
...
3:   aload_0
4:   getfield       VendingMachine.availType I (5)
7:   arraylength
8:   if_icmpge      #32
11:  aload_0
12:  getfield       VendingMachine.availType I (5)
...
18:  getfield       VendingMachine.Type I (4)
21:  if_icmpne      #26
24:  iconst_1
25:  ireturn
26:  iinc           %1    1
29:  goto           #2
32:  iconst_0
33:  ireturn
```

Fig.4: *The Bytecode Instructions for Vending Machine*

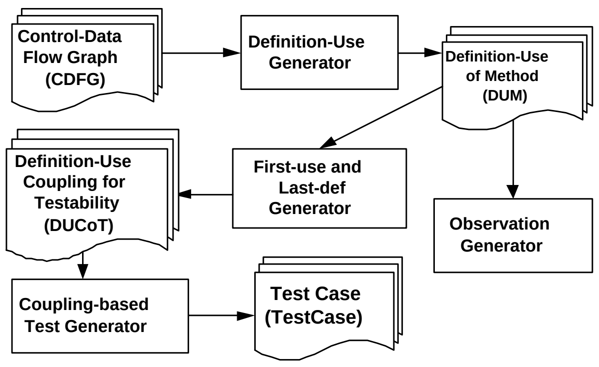

Control-Data Flow Graph (CDFG)
Definition-Use Generator
Definition-Use of Method (DUM)
Definition-Use Coupling for Testability (DUCoT)
First-use and Last-def Generator
Observation Generator
Coupling-based Test Generator
Test Case (TestCase)

Fig.6: *The Process for Increasing Class-component Testability*

the class (constant pool, fields, methods etc.). The *CDFGen* consists of three parts: the *leader generator*, the *basic block generator* and the *flow generator.* The *leader generator* generates the *Leader Hashtable* that will be used by the *basic block generator* and the *flow generator.* The *Leader Hashtable* contains leaders that were mentioned in section 3.1.1. The *basic block generator* divides bytecode instructions into basic blocks. The *getfield* and *putfield* instructions are also collected for each basic block. The *flow generator* generates flows of control between basic blocks. When the three processes of *CDFGen* finish, a *CDFG* has been created for a method. The following subsection explains how the *CDFG*s are used to increase testability of class-components.

3.2 Increasing Class-Component Testability

Testing software is easier when testability is high and, in general, increasing testability makes detecting faults easier. This subsection explains the process used to increase class-component testability. This process uses the *CDFG*s from the previous step to collect definition-use pairs where the definitions and uses are in different methods. The analysis is based on coupling relationships, and yields *Definition-Use Couplings for Testability* (*DUCoT*). The *DUCoT*s are used to control inputs (*increasing controllability*) and observe outputs (*increasing observability*) for a class-component. The process for increasing class-component testability is shown in Fig. 6.

3.2.1 Definition-Use Coupling for Testability (DUCoT)

The previous process constructs the *Control-Data Flow Graph* (*CDFG*) of a method to model the flow of control and data through that method. This process uses the *CDFG*s of a class-component to gather the definition and use information for all variables of each method. The first uses and the last definitions are also identified for each variable. The information gathered for a method is collectively called the *Definition-Uses of Method* (*DUM*). To collect definitions and uses for all variables of a method, the associated *CDFG* of a method is traversed from the first basic block (basic block 0) to the last basic block (*EXIT* block). To collect the first uses, the *CDFG* is traversed by starting from the first basic block and then

Table 1: *The CDFGs of Vending Machine*

Basic Block Number	**First-Inst**	**Last-Inst**	**DefUse**	**Pre-Block**	**Suc-Block**
Method <init>( )					
0	0	39	(d, total, 6), (d, curQtr, 11), (d, Type, 16), (d, availType, 36)		1
1	*EXIT*			0	
Method addQtr( )					
0	0	10	(u, curQtr, 2), (d, curQtr, 7)		1
1	*EXIT*			0	
Method returnQtr( )					
0	0	5	(d, curQtr, 2)		1
1	*EXIT*			0	
Method Vend( )					
0	0	14	(d, Type, 7), (u, curQtr, 11)		1, 2
1	17	25		0	9
2	28	33	(u, Type, 29)	0	3, 4
3	36	44		2	9
4	47	51		2	5, 6
5	54	62		4	9
6	65	70	(u, curQtr, 66)	4	7, 8
7	73	81		6	9
8	84	109	(u, total, 94), (d, total, 99), (u, curQtr, 104), (d, curQtr, 109)	6	9
9	112	140	(u, curQtr, 128)	1, 3, 5, 7, 8	10
10	*EXIT*			9	
Method available( )					
0	0	1			1
1	2	8	(u, availType, 4)	0, 4	2, 5
2	11	21	(u, availType, 12), (u, Type, 18)	1	3, 4
3	24	25		2	6
4	26	29		1	2
5	32	33		1	6
6	*EXIT*			3, 5	

following with each successor block, in a depth first manner. To collect the last definitions, the *CDFG* is traversed by starting from the last basic block and then following backward through predecessor blocks, again in a depth first manner. As shown in Fig. 6, the *Definition-use Generator* collects the *DUM*s for each method. The *DUM*s are used to collect definition-use pairs of a variable between the last definitions in a method and the first uses in other methods. The *First-use and Last-def Generator* in Fig. 6 collects this information. The definition-use pairs of a variable are called *Definition-Use Couplings for Testing (DUCoT).* For example, for the variable *total* from the vending machine in Table 1, the last definitions are in method <*init*> at position 6 and method *vend* at position 99, and the first use is in method *vend* at position 94. The remainder of this paper refers to the last definition at position x and the first use at the position y as the *last-def location* x and the *first-use location* y. Fig. 7 shows the *DUCoT*s for variables of the vending machine. A *DUCoT* is defined as follows:

Definition The *DUCoT* of variable v is a tuple,

$DUCoT\ (v) = (D_L, U_F)$

• D_L is a finite set of last definitions of variable v
Each element of D_L is M_d [L_d], where
M_d is a method that defines variable v, and
L_d is a last-def location in M_d
where variable v is defined

• U_F is a finite set of first uses of variable v
Each element of U_F is M_u [L_u], where
M_u is a method that uses a variable v, and
L_u is a first-use location in M_u
where variable v is used

3.2.2 Increasing Controllability

In this step, the *DUCoT*s are used to increase controllability by generating test inputs. Test inputs are generated according to the coupling-based criteria proposed by Jin and Offutt [18], as defined in Section 2.3. A testing criterion is a rule that im-

DUCoT (curQtr) = ({<init> [11], addQtr [7], returnQtr [2], vend [109]}, {addQtr [2], vend [11]})

DUCoT (total) = ({<init> [6], vend [99]}, {vend [94]})

DUCoT (Type) = ({<init> [16], vend [7]}, {vend [29], available [18]})

DUCoT (availType) = ({<init> [36]}, {available [4]})

Fig.7: *The DUCoTs of Vending Machine's Variables*

poses requirements on a set of test cases. Applying coupling-based testing to component testing requires some minor modifications to the terminology.

The *All-coupling-uses* criterion requires that for each coupling-def, at least one test case executes a path from the def to each reachable coupling-use. An adaptation of the standard All-coupling-uses for a component is given in the following definition.

Definition Let (D_L, U_F) be a *DUCoT* of variable v. The *All-coupling-uses* of variable v is defined as

$$\mathbf{AllCoU}\ (\boldsymbol{v}) = \{\ (\mathbf{M}_d[\mathbf{L}_d],\mathbf{M}_u[\mathbf{L}_u])\ |\ \forall \mathbf{M}_d[\mathbf{L}_d] \in \mathbf{D}_L \text{ and } \forall \mathbf{M}_u[\mathbf{L}_u] \in \mathbf{U}_F)\}$$

The test for an ordered pair ($M_d[L_d]$, $M_u[L_u]$) of variable v requires the path to execute from the last-def location L_d of method M_d to the first-use location L_u of method M_u without any intervening redefinitions of the variable v. The test requirements of All-coupling-uses of vending machine are shown in Table 2. As shown in Fig. 6, the *Coupling-based Test Generator* uses *DUCoT*s to generate test cases that satisfy the requirements.

The next step is to generate test data that satisfy all test requirements. A test case is a sequence of method calls. Test case i for variable v, ($M_d[L_d]$, $M_u[L_u]$), causes the execution to reach two specific locations. The first is the last-def location L_d of method M_d, which defines a variable v (*required def location*). The second is the first-use location L_u of method M_u, which uses a variable v (*required use location*). Moreover, this execution must not redefine variable v between these two locations, that is, this must be a *def-clear path execution.* The test data generation process is shown in Fig. 8. We use instrumentation at the bytecode level to detect the required def location and the required use location, and to check that the execution is def-clear. To keep track of the reached locations, the *Definition-Use Track Instrumentation* instruments the java .class file to produce the instrumented class (*Instrumented Class*). The instrumentation is performed by inserting instructions at each *putfield* and *getfield* instruction. When the inserted instructions are executed, they update and record the sequence of reached locations.

The test data generation process for test case i for variable v, ($M_d[L_d]$, $M_u[L_u]$), is as follows:

1. Generate an initial test case that consists of a class-component constructor, a call to method M_d and a call to method M_u. The initial test input data is generated randomly and set to be "temporary."
2. Execute the instrumented class with the temporary test data.
3. Evaluate the execution results. If (1) the sequence of locations executed contains the required def location L_d in method M_d, (2) the sequence of locations executed contains the required use location L_u in method M_u after L_d in method M_d, and (3) there is no other definition of v between location L_d and location L_u, the temporary test data is set to be the relevant test data for test case i. Otherwise, the temporary test data is modified by adding a method call or changing the values of the method call arguments.
4. If the *DUCoT* was not covered, go to step 2, else, exit.

The result of this process is the sequence of method calls that execute a coupling path from the required coupling-def to the required coupling-use. The case study in the next section illustrates how to use these ideas to increase controllability.

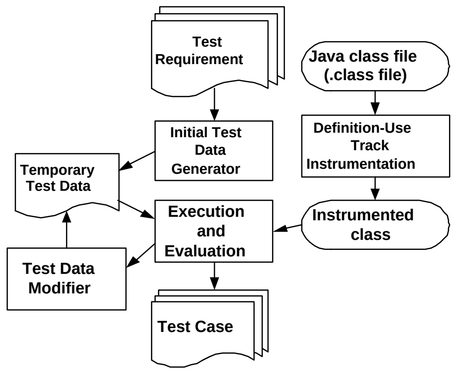


Fig.8: *The DUCoTs of Vending Machine's Variables*

Table 2: *The All-coupling-uses of Vending Machine's Variables*

#	**Variable**	**All-coupling-uses**
1	curQtr	<init> [11] , addQtr[2]
2		<init> [11], vend[11]
3		addQtr [7], addQtr[2]
4		addQtr [7], vend[11]
5		returnQtr [2], addQtr[2]
6		returnQtr [2], vend[11]
7		vend [109], addQtr[2]
8		vend [109], vend [11]
9	total	<init> [6], vend [94]
10		vend [99], vend [94]
11	Type	<init> [16], vend [29]
12		<init> [16], available [18]
13		vend [7], vend [29]
14		vend [7], available [18]
15	availType	<init> [36], available [4]

3.2.3 Increasing Observability

Binder [7] makes the following point about observability in object-oriented software: "if users cannot observe the output, they cannot be sure how a given input has been processed." Observability focuses on the ease of observing outputs. Observability requires the test engineer to be able to determine if the software behaves correctly during testing. The observability of black-box software is inherently limited because only the outputs are visible. Although encapsulation and data hiding brings many benefits, it perversely causes problems with testing by making object-oriented software less observable [26]. The internal state is not readily available and it does not always have a direct impact on the output. This makes it possible for the internal state to be erroneous but the final output to be correct. Although this may not be an immediate problem for the current software (the output is, after all, correct), incorrect internal states represent potential problems that can cause major failures in the future, particularly if the software evolves. Therefore, access to the internal state is crucial to effective testing.

The most common way to increase observability is debugging. Debuggers view all state information at a specific point in execution.

Observability is reduced as the size of the program increases. To solve this problem, we introduce observation points, which we call *observation probes*, to observe the relevant internal state variables of executions during testing. The DUCoTs from the previous step are used to establish observation probes and monitor associated state variables. Observation probes should be inserted **before first-use locations**. Consider the fourth All-coupling-use of variable *curQtr* in table 2, (addQtr [7], vend[11]). This refers to the last-def location 7 in method *addQtr* and the first-use location 11 in method *vend*. Following section 3.2.2, the test data for this case consists of calls to methods *addQtr* and *vend*. The last-def and first-use are reached by executing the methods *addQtr* and *vend*. Therefore, the observation probe is inserted before the method call *vend* to observe the value of variable *curQtr*. The case study in the next section illustrates how to use these ideas to increase observability.

4. CASE STUDY

This section describes a case study that was carried out to demonstrate our ideas. To show the effectiveness of the process for increasing class-component testability, we evaluated how easily faults can be revealed and observed. As discussed in the previous section, class-component testability is increased by separately increasing controllability and observability. To increase controllability, we developed a method to generate test cases based on the All-coupling-uses test criterion. To increase observability, the observation probes were added to show current values of internal state variables.

Mutation is widely considered to be one of the strongest testing techniques, and is often used as a "gold standard" against which to evaluate other testing techniques. It has been used as a way to induce faults into the program for empirical fault studies in dozens of testing papers [3, 9, 10]. Andrews et al. [3] recently studied the direct question of whether mutation-like faults are valid in studies like this, and found that they are. This supports older evidence [23], which found that tests that detect mutation-like faults are good at detecting more complicated faults.

Mutation analysis works by modifying copies of the original program or component. Mutants are created by making copies of the original version, then inducing each mutant change into a unique copy. These faulty copies are called *mutants*. *Mutation operators* define rules for how to change the original program. Examples include changing arithmetic operators, logic operators, and variable references. Tests are run on the original component, and then each mutated copy. The results from the original version are compared with the results from the mutated versions to see if the tests were able to find the mutants. After this testing, the mutants are thrown away.

This work used the testing tool Jester [22] to generate mutants. Tests generated to satisfy All-coupling-uses were compared with tests generated to cover every statement. The comparison was carried out in terms of how many mutants were detected by each set of tests.

Our experiment proceeded in six steps: (1) prepare classes to test, (2) generate sets of tests following our approach for each class, (3) generate sets of tests to satisfy statement coverage for each class, (4) generate the mutants for each class, (5) run each set of tests on the original and each mutated version, and (6) count the tests that caused the program to fail and compute the fault detection ability of each test set.

We used five subjects, the vending machine class and four classes from a data structure package, LinkedList, StackAr, QueueAr and BinarySearchTree. Following our proposed method, test cases were generated to satisfy All-coupling-uses for each class. These are called *All-coupling-uses tests* (*ACU* tests). The ACU tests were duplicated, and observations of internal states were added. These are called *All-coupling-uses with observability tests* (ACU-O tests). Also, test cases for each of classes were generated to satisfy the statement coverage criterion (*SC* tests). Each test case is a sequence of method calls and was prepared as JUnit tests for automated execution [5].

The JUnit test cases are used by Jester [22]. We use Jester to generate mutants for each of classes. The ACU-O tests are the same as the ACU tests, but with JUnit assert methods added. These assertions

Table 3: *Case Study Results on All-coupling-uses and All-coupling-uses with Observability*

Class Name	**#Tests**	**#Mutants**	**ACU**		**ACU-O**		**Detection Increase**
			Faults Found	**% Faults Found**	**Faults Found**	**% Faults Found**	
VendingMachine	14	27	**1**	4	**16**	59	16.0
LinkedList	24	10	**5**	50	**8**	80	1.6
StackAr	29	18	**5**	28	**11**	61	2.2
QueueAr	22	20	**3**	15	**14**	70	4.7
BinarySearchTree	61	49	**18**	38	**30**	61	1.7
Sum	130	124	**32**		**79**		

Table 4: *Case Study Results on Statement Coverage and All-coupling-uses with Observability*

Class Name	**#Mutants**	**SC**			**ACU-O**			Detection Increase
		#Tests	**Faults Found**	**% Faults Found**	**#Tests**	**Faults Found**	**% Faults Found**	
VendingMachine	27	7	**1**	4	14	**16**	59	16.0
LinkedList	10	11	**4**	40	24	**8**	80	2.0
StackAr	18	9	**8**	45	29	**11**	61	1.4
QueueAr	20	7	**3**	15	22	**14**	70	4.7
BinarySearchTree	49	21	**17**	35	61	**30**	61	1.8
Sum	124	55	**30**		130	**79**		

were observation probes to check internal states during testing, and thus enhance observability. All mutants were executed by all three sets of tests, ACU, ACU-O and SC. Then, fault detection scores were computed in terms of the number of faults found. Note that this process to increase testability does not use source code. The only use of the source in this study was by Jester to generate the mutants.

Table 3 shows the fault detection ability of the ACU and ACU-O tests. The ACU-O tests found more faults than the ACU tests on all classes. The average difference in fault detection is 2.5.

Table 4 shows the results of executing the mutants with SC tests and ACU-O tests. ACU-O tests found more faults than SC tests for all classes. The fault detection ability of the ACU-O was 2.6 times more than that of the SC tests.

5. CONCLUSIONS

This paper described an analysis technique to increase testability of class-components that operates at the bytecode level. The analysis gathers control and data flow information and automatically creates an intermediate graph, CDFG, for each method in class-components. CDFGs are used with coupling-based test criteria to supply inputs for integration testing of class-components, and to make it easier to observe internal state variables during testing to increase the detection of failures. Our approach focuses on intra-class method calls, and does not look for problems that exist in inheritance and polymorphism.

Component-based (CB) software development is currently in widespread use. A CB system is built by assembling already existing components, which need to be retested in the new environment. This technique can be used to increase the testability of reused components whose source is not available.

A common problem when testing class-components that use information hiding is that erroneous states in the software may be “masked” or “hidden,” that is, the erroneous states may not result in a failure that can be observed externally. This may cause the underlying flaw to be left in the software; leading to failures during use or after the software is later modified. The method in this paper to increase observability can allow more faults to be revealed during testing. The case study indicated that our technique can help to increase the detection of failures. The results show that our method offers significantly more fault detection than statement coverage. For practical use of these techniques, they must be fully automated in a tool that analyzes the classes and generates values to satisfy the criterion. We are currently implementing this tool.

ACKNOWLEDGMENTS

This work was supported in part by Thailand’s Commission of Higher Education (MOE), and by the Center of Excellence in Software Engineering, Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University. Thanks to the Department of Information and Software Engineering, School of Information Technology & Engineering at George Mason University, for hosting the first author during parts of this research project.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Reading, MA, 1986.

[2] R. T. Alexander and A. J. Offutt, "Analysis Techniques for Testing Polymorphic Relationships," *Proceedings of the Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS30 '99)*, 1999, IEEE Computer Society, Santa Barbara CA, pp.104-114.

[3] J. H. Andrews, L. C. Briand and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," *Proceedings of 27th International Conference on Software Engineering*, 2005, St. Louis Missouri, USA, pp. 402-411.

[4] Atanas Neshkov, "NavExpress DJ Java Decompiler," Available on-line at: *http://www.dj.navexpress.com* (last access August 2006).

[5] K. Beck, and E. Gamma, "JUnit A Cook's tour," Available on-line at: *http://junit.sourceforge.net/doc/cookstour /cookstour.htm* (last access August 2006).

[6] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990.

[7] R. V. Binder, "Design for testability with object-oriented systems," *Communications of the ACM*, Vol.37, No.9, pp.87-101, 1994.

[8] M. Dahm, "Byte code engineering with the JavaClass API," *Technical Report B-17-98*, Freie Universität Berlin, Institut für Informatik, January 1999.

[9] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, Vol.11, No.4, pp.34-41, April 1978.

[10] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, Vol.17, No.9, pp.900-910, September 1991.

[11] R. Freedman, "Testability of software components," *IEEE Transactions on Software Engineering*, Vol.17, No. 6, pp.553–563, June 1991.

[12] L. Gallagher, J. Offutt and A. Cincotta, "Integration testing of object-oriented components using finite state machines," *The Journal of Software Testing, Verification, and Reliability*, In press -published online January 2006.

[13] M.J. Harrold, A. Orso, D. Rosenblum, G. Rothermel, M.L. Soffa and H. Do, "Using component metadata to support the regression testing of component-based software," *Proceedings of IEEE International Conference on Software Maintenance*, 2001, Florence, Italy, pp.154-163.

[14] M. J. Harrold and G. Rothermel, "Peforming data flow testing on classes," *Proceeding on ACM SIGSOFT Foundation of Software Engineering*, 1994, New Orleans, LA, pp.154–163.

[15] M. J. Harrold and M. L. Souffa, "An incremental approach to unit testing during maintenance," *Proceedings of IEEE/ACM Conference on Software Maintenance*, 1988, Phoenix, Arizona, USA, pp.362-367.

[16] H. S. Hong, Y. R. Kwon and S. D. Cha, "Testing of Object-Oriented Programs Based on Finite State Machines," *Proceedings of Asia-Pacific Software Engineering Conference (ASPEC95)*, 1995, Brisbane, Australia, pp. 234-241.

[17] IEEE Standard Glossary of Software Engineering Technology, ANSI/IEEE 610.12, *IEEE Press* (1990).

[18] Z. Jin and J. Offutt, "Coupling-based criteria for integration testing," *The Journal of Software Testing, Verification, and Reliability*, Vol.8, No.3, pp.133–154, September 1998.

[19] S. Kansomkeat, J. Offutt and W. Rivepiboon, "Increasing class-component testability," *Proceedings of The IASTED International Conference on Software Engineering*, 2005, Innsbruck, Austria, pp.156-161.

[20] J. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Transactions on Software Engineering*, Vol.9, No.3, pp.347-354, May 1983.

[21] T. Lindholm and F. Yellin, *The Java^{TM} Virtual Machine Specification*, Addison-Wesley, second edition, 1999.

[22] I. Moore, "Jester – a JUnit test tester," *Proceedings of the 2nd InternationalConference on Extreme Programming and FlexibleProcesses in Software Engineering*, 2001, Sardinia, Italy, pp. 84-87.

[23] J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering Methodology*, Vol.1, No.1, pp.3-18, January 1992.

[24] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, Vol.11, No.4, pp.367-375, April 1985.

[25] B.-Y. Tsai, S. Stobart and N. Parrington, "Employing data flow testing on object-oriented classes," *The IEE Proceedings – Software*, Vol.148, No. 2, pp.56-64, 2001.

[26] J.M. Voas and K.W. Miller, "Software testability: The new verification," *IEEE Software*, Vol.12, No.3, pp.17-28, May 1995.

[27] Y. Wang, G. King, M. Fayad, D. Patel, I. Court, G. Staples, and M. Ross, "On Built-in Test Reuse in Object-Oriented Framework Design," *ACM Journal on Computing Surveys*, Vol. 32, No. 1, March 2000.

[28] E. Weyuker, "Testing Component-based Software: A Cautionary Tale," *IEEE Software*, Vol. 15, No. 5, pp. 54-59, Sept/Oct 1998.

Supaporn Kansomkeat received the B.S. (Mathematics) degree in 1991 and M.S. (Computer Science) in 1995 from Prince of Songkla University. She is currently pursuing the PhD degree in Computer Engineering at Chulalongkorn University, and joined the Software Engineering Laboratory in 2002. Her research is concerned with software testing and increasing component testability.

Jeff Offutt is a Professor of Information and Software Engineering at George Mason University. His current research interests include software testing, analysis and testing of web applications, object-oriented program analysis, module and integration testing, formal methods, and software maintenance. He has published over 100 research papers in refereed software engineering journals and conferences. Offutt is editor-in-chief of Wiley's journal of Software Testing, Verification and Reliability, is or has been on the editorial boards for the IEEE Transactions on Software Engineering (2001-2005), the Empirical Software Engineering Journal (current), the Journal of Software and Systems Modeling (current), and the Software Quality Journal (current) and was program chair for ICECCS 2001. He received the Best Teacher Award from the School of Information Technology and Engineering in 2003. Offutt received a PhD degree in Computer Science from the Georgia Institute of Technology, and is a member of the ACM and IEEE Computer Society.

Wanchai Rivepiboon is an associate Professor of Department of Computer Engineering, Chulalongkorn University. He received a PhD degree in Computer Science from University Grenoble 1. His current research interests include Artificial Intelligence and Software Engineering. He is currently head of the Software Engineering Laboratory in the Department of Computer Engineering, Chulalongkorn University.