SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2008; **18**:193–219
Published online 3 March 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.387

An analysis technique to increase testability of object-oriented components

Supaporn Kansomkeat*,† and Wanchai Rivepiboon

Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok 10330, Thailand

SUMMARY

Object-oriented component engineering is increasingly used for system development, partly because it emphasizes portability and reusability. Each time a component is used, it must be retested in the new environment. Unfortunately, the data abstraction that components usually use results in low testability. First, internal variables cannot be directly set. Second, even though a test input may trigger a fault, the failure does not propagate to the output. This paper presents a technique to increase object-oriented component testability, thereby making it easier to detect faults. Components are often sealed so that source code is not available. The program analysis is performed at the Java component bytecode level. A component's bytecode is analysed to create a *control and data flow graph*, which is then used to increase component testability by increasing both controllability and observability. We have implemented this technique and applied it to several components. Experimental results reveal that fault detection can be increased by using our increasing testability process. Copyright © 2008 John Wiley & Sons, Ltd.

Received 17 August 2007; Revised 12 December 2007; Accepted 14 December 2007

KEY WORDS: software testing; software testability; component software; bytecode; program analysis

1. INTRODUCTION

Software testing is used to validate software quality and reliability, but it can be expensive and labour-intensive [1]. Software testing attempts to reveal software faults by executing the program on inputs and comparing the outputs of the execution with expected outputs. A key factor in the

*Correspondence to: Supaporn Kansomkeat, Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok 10330, Thailand.
†E-mail: supaporn.k@student.chula.ac.th, supaporn.k@psu.ac.th

Contract/grant sponsor: Thailand's Commission of Higher Education (MOE)
Contract/grant sponsor: Center of Excellence in Software Engineering, Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University

field of software testing is the ability to reduce the test effort [2–4]. 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 [5], *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 [6] focus software testability 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 terms of controllability and observability. *Controllability* is the probability that users are able to *control* a component's inputs (and internal state). *Observability* is the probability that users are able to *observe* a component's outputs. Freedman [8] also considered testability based on the notions of observability and controllability. Observability captures the degree to which a component can be observed to generate the correct output for a given input. Controllability refers to the ease of producing all values of its specified output domain.

Although these definitions are different, they are consistent in that they all view component testability generally as how easy it is to test the software. High testability means that existing faults can be revealed relatively easily during testing, inputs can be easily selected to satisfy testing criteria, and outputs of state variables can be observed during testing. A component with good testability is important because test tasks are easier and test costs are reduced. Voas and Miller [6] 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.* [9] increase component testability by putting complete test cases inside the components, called the built-in test (BIT) approach. The tests are constantly present and reused with the component. One disadvantage of the BIT approach is the growth of programming overhead and component complexity. Another is that component developers cannot be expected to provide BITs or testing information to component users. Instead of increasing testability by using BIT, this paper tests components as they are integrated into new contexts This is a form of integration testing that asks whether the components behave appropriately in this new context. The goal of development organizations that we work with finds behavioural mismatches to be a major source of failure in integration testing, system testing, and deployment.

Object-oriented (OO) component engineering is increasingly used for system development, partly because it emphasizes portability and reusability. Each time a component is used, it must be retested in the new environment. Weyuker [10] suggests that a component should be tested many times individually, and also each time it is integrated into a new system. This paper presents a model to increase the testability of an OO component when it is reused. Some information about a component is needed to increase its testability, such as information obtained through program analysis. Generally, program analysis is used to inspect programs to gather properties such as control and data flow information. The previously written components are often sealed so that source code is not available. This makes program analysis significantly more difficult. This paper addresses this problem by performing program analysis at the bytecode level.

This paper presents an analysis technique to analyse an OO component at the bytecode level that is used to directly increase component testability without requiring access to the source. First, a component's bytecode is analysed to create a *control and data flow graph* (CDFG), which is then used to obtain definition and use information of methods and class variables in the component. Then, the definition and use information is used to increase component testability by increasing both controllability and observability. Controllability makes it easier to generate tests to exercise an OO component in various ways; thus, faults can more easily be revealed. Observability helps to monitor the results of testing; thus, OO component failures can be detected more easily. A previous paper [11] introduced preliminary versions of these ideas. This paper presents the bytecode-based analysis process in detail and the process of constructing the intermediate form that represents definition and use information. This paper also takes the next step to increase observability by tracking and asserting relevant internal state variables and include substantially more experimental validation. This paper considers a Java class to be an OO component.

The remainder of this paper is organized as follows. Section 2 introduces the background. Section 3 presents a bytecode-based analysis process. Section 4 describes the process of increasing class-component testability. Then, a case study is shown in Section 5. Conclusions and future work are presented in Section 6.

2. BACKGROUND

This paper presents an analysis technique to increase OO component testability. The analysis is carried out at the bytecode level. The bytecode instructions are parsed to collect data flow information. This information provides ways to generate test inputs and observe the outputs of testing. Coupling-based criteria are used to guide test selection. The tests are evaluated by their fault detection ability based on mutation testing. This section provides a brief overview of these topics.

2.1. Java bytecode instructions

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. This file is dynamically loaded into an interpreter (Java Virtual Machine (JVM)) and executed. Figure 1 shows an example of a Java class, setCoeff, and the bytecode instructions for member method calCoeff.

The Java bytecode instructions can be roughly grouped as follows:

- *Stack operations*: Pushing constants onto the stack and retrieving them (e.g. *ldc*, *iconst_5*, *bipush*).
- *Arithmetic operations*: Computing a result as a function of values on the operand stack and pushing the result back on the operand stack. 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*: Calling methods (e.g. *invokestatic*, *invokevirtual*, *invokespecial*).
- *Return Instruction*: Returning methods (e.g. *return*, *ireturn*, *areturn*).

```
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
```

Figure 1. A simple Java class and the bytecode instructions for method CalCoeff.

- *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 [12].

In Java, an *exception* is an event that occurs during program execution 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*. The catch block in the left side of Figure 1 is the exception handler.

2.2. Data flow analysis

Data flow testing [13,14] tries to ensure that the correct values are stored into memory, and then that they are subsequently used correctly. A definition (*def*) is a statement where a value is stored into memory. A *use* is a statement where a value is accessed. A *definition–use pair* (or du-pair) of a variable is an ordered pair of a definition and a use, with the limitation that there must be an execution path from the definition to the use without any intervening redefinition of the variable (the path must be *def-clear*). Data flow criteria require tests to execute paths from specific definitions to uses by selecting particular definition–use pairs to test. Two data flow testing criteria were first defined by Laski and Korel [13]. They proposed the *all-definitions* criterion, which requires that tests should cover a path from each definition to at least one use, and the *all-uses* (AU) criterion, which requires tests to cover a path from each definition to *all reachable* uses.

2.3. Coupling-based testing

Data flow testing has been applied inter-procedurally in two different ways. Harrold and Rothermel [15] proposed a complete control and data flow analysis to compute the *program dependency graph* (which combines control and data flows). Directly applying either the all-defs or the AU criterion in this way is very expensive, both in terms of number of du-pairs and the difficulty in resolving the paths. Jin and Offutt [16] proposed a simpler model that just focuses on connections between pairs of methods. *Coupling-based testing* (CBT) applies data flow testing to the integration level by requiring the program to execute data transfers from definitions of variable in a caller to uses of the corresponding variables in the callee. Instead of all variables' definitions and uses, CBT is concerned only 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 [16] defined four coupling-based integration test coverage criteria. This paper uses the *all-coupling-uses* (ACU) criterion. ACU 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*.

2.4. Mutation testing

Mutation analysis [17,18] is often used to assess the adequacy of a test set. It is a fault-based testing strategy that starts with a program (original) to be tested and makes numerous small syntactic changes to the original program. Programs with injected faults are called *mutants*. Mutants are obtained by applying mutation operators that introduce the simple changes into the original program. For example, one operator changes relational operators:

A program segment *P*:

```
1. if (x < 10)
2.    doA( )
3. if (x < 20)
4.    doB( )
```

A mutation of P on line 1 would be

```
1. if (x > 10)
2.     doA( )
3. if (x < 20)
4.     doB( )
```

If a test set causes behavioural differences between the original program and the mutant, the mutant is considered to be *killed* by the test. The product of mutation analysis is a measure called *mutation score*, which indicates the percentage of mutants killed by a test set. The mutation score indicates the adequacy of the test cases. Some mutants are functionally *equivalent* to the original program; they always produce the same output as the original program and thus cannot be killed by any test cases. Equivalent mutants are not counted in the mutation score.

3. BYTECODE-BASED ANALYSIS

This paper is based on data flow testing that considers a flow between a variable definition and subsequent uses of that variable. The supporting information needed is control flow and data flow information.

The most common method to represent the control flow information is a *control flow graph* (*CFG*) [19]. Each node in a CFG represents a statement or a basic block of statements, and edges represent the flow of control between pairs of nodes. To represent both control and data flow information, this research extends the CFG by collecting variables defined and used in each node. The extended CFG to represent control and data flow information is called the *CDFG*.

Conventional program analysis collects control and data flow information from program source code. Because the source is not available in component-based software, this analysis technique gathers such information at the *bytecode* level. This paper considers a Java class to be an OO component.

Java bytecode is analysed to collect control and data flow information and results in a CDFG for each method. The CDFG includes both. The CDFG also includes exception handling control, which was not defined for the CFG. The exception handling control flow can be raised in the Java class component through a *throw* statement (for example, *throw* new ArithmeticException()). The CDFG of a method is used to obtain definition and use information. Table I shows the bytecode instructions used in the analysis process.

Table I. Bytecode instructions used in the analysis process.

Group	Bytecode instructions
Unconditional branch	goto
Conditional branch	if_acmpeq, if_acmpne, if_icmpeq, if_icmpge, if_icmpgt, if_icmple, if_icmplt, if_icmpne, ifeq, ifge, ifgt, ifle, iflt, ifne, ifnonnull, ifnull, lookupswitch, tableswitch
Return	areturn, dreturn, freturn, ireturn, lreturn, return
Athrow	athrow

3.1. CDFG

The CDFG construction process is as follows. First, the bytecode instructions are extracted and partitioned into *basic blocks*, and then the flows of control and data are added. A *basic block* is a sequence of consecutive bytecode instructions in which flow of control enters at the beginning and leaves at the end without halting or branching except at the end [19]. In a basic block, if any instruction is executed, all instructions will be executed. A basic block has only one entry point and one exit point. To create a basic block, *leader instructions*, instructions that begin basic block, are identified. A leader instruction can be:

- The first instruction.
- Any instruction that is the target of either a conditional branch group instruction or an unconditional instruction.
- Any instruction that immediately follows either a conditional branch group instruction or an unconditional instruction.
- Any instruction that immediately follows a return group instruction or *athrow* instruction (exception handling).
- The first instruction of an *exception handler*.

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. An *EXIT* block is added to be the exit point. For example, the bytecode instructions for method *calCoeff* in Figure 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 Figure 2. Each basic block is analysed to gather the definition and use information (*DefUse information*) for data flow analysis. DefUse information refers to variable definitions and uses. An instruction is considered to perform a *definition* of a variable (*def instruction*) if a value is stored into that variable from the operand stack. An instruction is considered to perform a *use* of a variable (*use instruction*) if its value is accessed and loaded onto the stack. Java bytecode contains two types of field variables, *instance fields* (non-static fields) and *class fields* (static fields). The instance fields are unique to each object of the class. The class fields are unique to the entire class. This paper focuses on *instance fields*, which are defined and used between methods; therefore, local variables will not be considered. Because of the complexity of array bytecode instructions, the work thus far has been limited to arrays of type int. Furthermore, the DefUse information of any element within an array is considered the DefUse information of the whole array. The reference Java bytecode instructions used to gather the DefUse information are shown in Table II.

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, depending on the type of the last instruction in B1, which could be one of the following five cases. Figure 3 illustrates these cases by showing the control flow between the basic blocks in calCoeff.

Case 1: *An instruction of an unconditional branch group*. 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 Figure 3, *Flow7* in Figure 3).

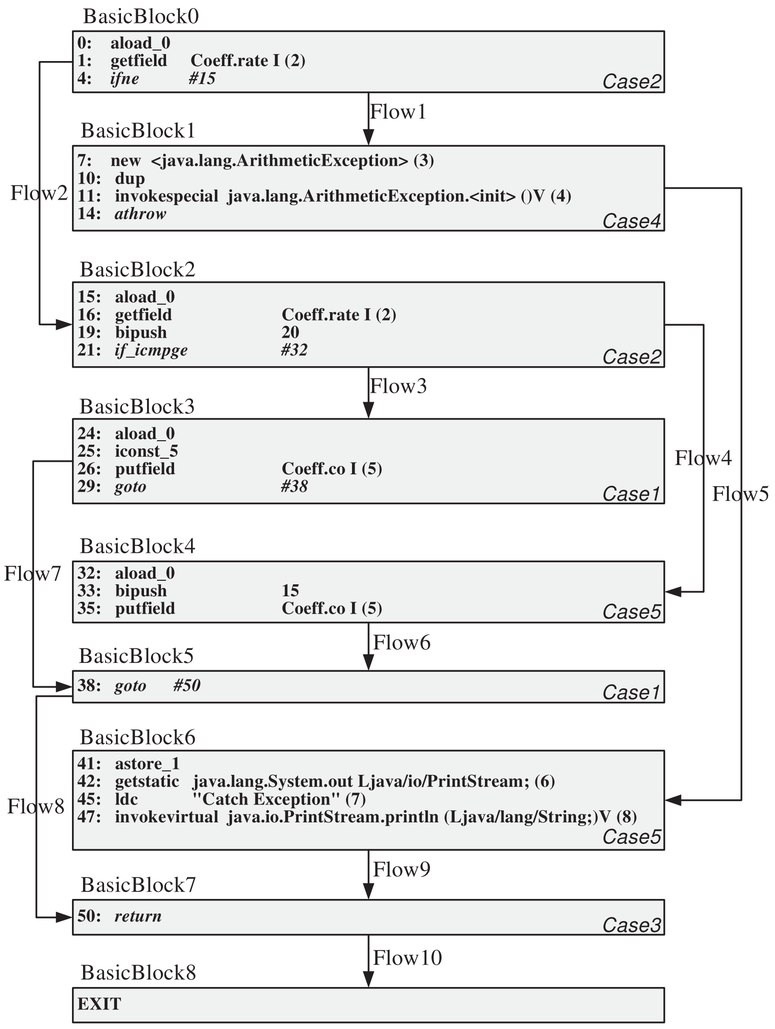
Case 2: *An instruction of a conditional branch group*. Two edges are added from B1. The first is to the basic block whose leader is the first instruction that directly follows the last instruction of B1 (e.g. from *BasicBlock0* to *BasicBlock1*, *Flow1*). The second is to the basic block

```
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> (3)
10:  dup
11:  invokespecial      java.lang.ArithmeticException.<init> ()V (4)
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
```

Figure 2. Partitioning method calCoeff in Figure 1 into basic blocks.

Table II. DefUse information.

Bytecode instructions	Description	Def/use instruction
getfield	Load a value of an instance field onto the operand stack	Use
putfield	Store a value from the operand stack as an instance field	Def
iaload	Load an array component onto the operand stack	Use
iastore	Store a value from the operand stack as an array component	Def



```
BasicBlock0
0:   aload_0
1:   getfield     Coeff.rate I (2)
4:   ifne         #15
                                                    Case2

BasicBlock1
7:   new  <java.lang.ArithmeticException> (3)
10:  dup
11:  invokespecial  java.lang.ArithmeticException.<init> ()V (4)
14:  athrow
                                                    Case4

BasicBlock2
15:  aload_0
16:  getfield                Coeff.rate I (2)
19:  bipush                  20
21:  if_icmpge               #32
                                                    Case2

BasicBlock3
24:  aload_0
25:  iconst_5
26:  putfield                Coeff.co I (5)
29:  goto                    #38
                                                    Case1

BasicBlock4
32:  aload_0
33:  bipush                  15
35:  putfield                Coeff.co I (5)
                                                    Case5

BasicBlock5
38:  goto     #50
                                                    Case1

BasicBlock6
41:  astore_1
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)
                                                    Case5

BasicBlock7
50:  return
                                                    Case3

BasicBlock8
EXIT
```

Figure 3. The flows between basic blocks of method calCoeff.

whose leader is the target of the branch instruction of B1 (e.g. from *BasicBlock0* to *BasicBlock2*, *Flow2*).

Case 3: *An instruction of a return group*. An edge is added from B1 to the exit point, *EXIT* (e.g. from *BasicBlock7* to the *EXIT* block, *Flow10*).

Case 4: *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*, *Flow5*).

*C*ase 5: *Not an instruction of branch*, *return or athrow group*. This happens when 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*, *Flow6*).

We illustrate our technique with a vending machine example. The Java source code and bytecode instructions for the vending machine are shown in Figures 4 and 5. The numbers preceding the line in Figure 5 indicate the instruction positions of the bytecode. The '. . . ' indicates omitted instructions. Figure 6 depicts the CDFGs of each method of vending machine. In this figure, basic blocks are represented as compartmentalized rectangles. The top compartment contains the basic block number, the middle compartment contains the first and last instruction positions of the basic block, and the bottom compartment contains the sequence of DefUse information. Each element of this sequence 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. An arrow shows the flow of control between basic blocks. A *predecessor block* of a current block is the basic block that has the flows of control to the current block. A *successor block* of current block is the basic block that has the incoming flows of control from the current block.

Data flow analysis obtains relationships among variables from flow graphs. This technique examines definitions and the subsequent uses of variables. Suppose instruction I_1 assigns a value to x, which instruction I_2 then uses. Then, instructions I_1 and I_2 have a data flow relationship. Using DefUse information from the previous step, data flow analysis can be processed by traversing the basic blocks in the CDFGs. 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*.

3.2. Definition and use information

The previous process constructs the *CDFG* of a method to model the flow of control and data through that method. This process uses the CDFGs of an OO component to collect the definition and use information. The process collects (1) the definition and use information for all variables of each method and (2) the first-use and the last-definition information. The information gathered for a method is collectively called the *definition and uses information* (*DUI*). To collect definition and use information for all variables of a method, the analysis procedure traverses a method's CDFG from the first basic block (basic block 0) to the last basic block (*EXIT* block). To collect the first-uses, the procedure starts from the first basic block and then follows each successor block in a depth first manner. To collect the last-definitions, the procedure traverses from the last basic block backward through predecessor blocks, again in a depth first manner. Table III summarize the DUIs of vending machine. Columns use and def show the sequence of use and definition instruction positions of variables. Column first-use and last-def show the sequence of first-use and last-definition instruction positions of variables.

3.3. Overview of methodology

An analysis tool was implemented to analyse automatically Java class files and generate CDFGs and DUIs. The tool was implemented by using an open-source tool from Apache/Jakarta [20],

```
class VendingMachine {

  private int total = 0;
  private int curQtr = 0;
  private int type = 0;
  private int availType = 2;

  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( ) {
     if (availType == type)
            return true;
    else
           return false;
  }
 } // class VendingMachine
```

Figure 4. The vending machine class.

```
void <init>()
0:   aload_0
…
6:   putfield        VendingMachine.total I (2)
…
11:  putfield        VendingMachine.curQtr I (3)
…
16:  putfield        VendingMachine.type I (4)
…
21:  putfield        VendingMachine.availType I (5)
24:  return

void addQtr()
0:   aload_0
1:   aload_0
2:   getfield        VendingMachine.curQtr I (3)
…
7:   putfield        VendingMachine.curQtr I (3)
10:  return

void returnQtr()
0:   aload_0
1:   iconst_0
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
17:  getstatic       java.lang.System.out
…
25:  goto            #112
28:  aload_0
29:  getfield        VendingMachine.type I (4)
32:  iload_2
33:  if_icmple       #47
36:  getstatic       java.lang.System.out
…
44:  goto            #112
47:  aload_0
48:  invokevirtual   VendingMachine.available ()Z (10)
51:  ifne            #65
54:  getstatic       java.lang.System.out
…
62:  goto            #112
65:  aload_0
66:  getfield        VendingMachine.curQtr I (3)
69:  iload_3
70:  if_icmpge       #84
73:  getstatic       java.lang.System.
…
81:  goto            #112
84:  getstatic       java.lang.System.out
…
94:  getfield        VendingMachine.total I (2)
…
99:  putfield        VendingMachine.total I (2)
…
104: getfield        VendingMachine.curQtr I (3)
…
109: putfield        VendingMachine.curQtr I (3)
112: getstatic       java.lang.System.out
…
128: getfield        VendingMachine.curQtr I (3)
…
140: return

boolean available()
0:   aload_0
1:   getfield        VendingMachine.availType I (5)
4:   aload_0
5:   getfield        VendingMachine.type I (4)
8:   if_icmpne       #13
11:  iconst_1
12:  ireturn
13:  iconst_0
33:  ireturn
```

Figure 5. The bytecode instructions for vending machine.

a bytecode manipulation library called BCEL (Byte Code Engineering Library). The BCEL API can help to analyse, create, and manipulate Java bytecode files.

Figure 7 illustrates the bytecode-based analysis process. The process consists of three major components: (1) the *class parser*, (2) the *control and data flow graph generator* (*CDFGen*), and (3) the *definition and use information generator*. The *class parser* parses the Java .class file and creates the *JavaClass* object, which represents all the information about the class (constant pool,

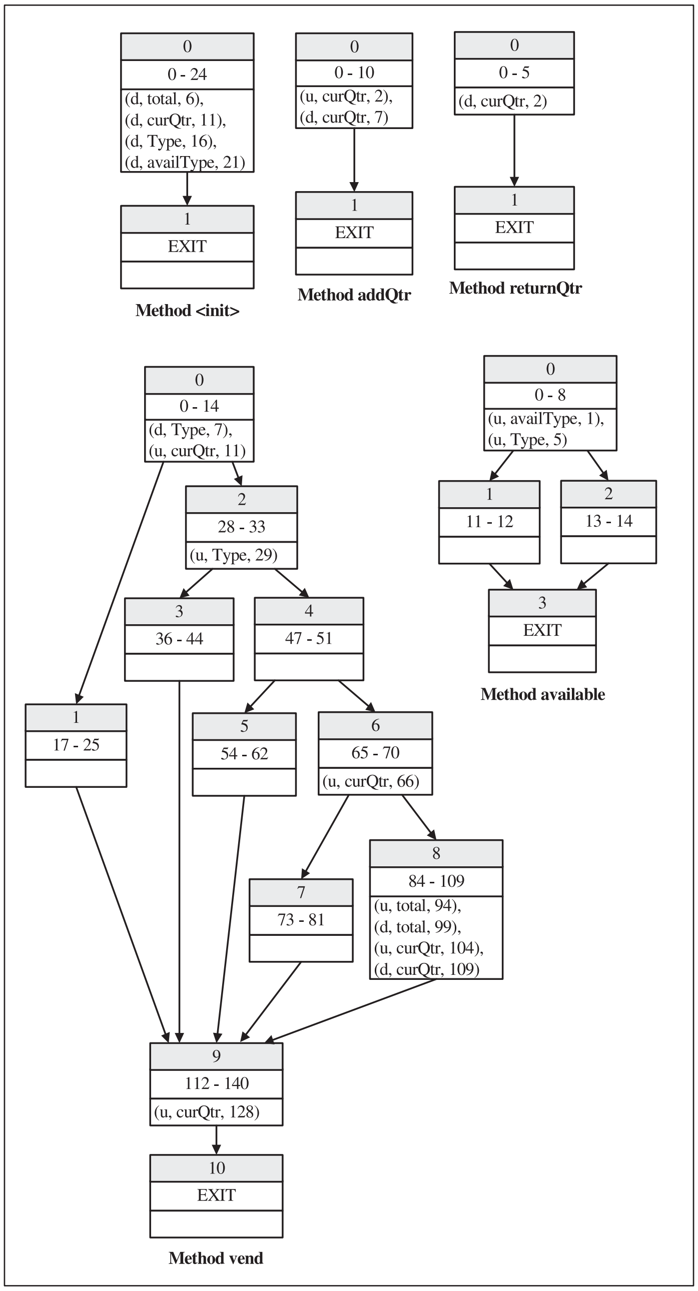


Figure 6. The CDFGs of each method of vending machine.

Table III. The DUIs of vending machine.

		Positions			
Method name	Variable name	Use	Def	First-use	Last-def
⟨init⟩ ( )	curQtr		11		11
	total		6		6
	type		16		16
	availType		21		21
addQtr ( )	curQtr	2	7	2	7
returnQtr ( )	curQtr		2		2
Vend ( )	curQtr	11, 66, 104, 128	109	11	109
	total	94	99	94	99
	type	29	7	29	7
available ( )	type	5		5	
	availType	1		1	

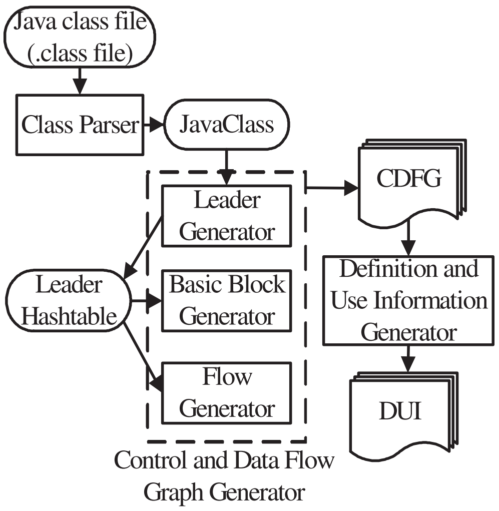

Java class file (.class file)
Class Parser
JavaClass
Leader Generator
Leader Hashtable
Basic Block Generator
Flow Generator
Control and Data Flow Graph Generator
CDFG
Definition and Use Information Generator
DUI

Figure 7. Bytecode-based analysis process.

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 the leaders that were described in Section 3.1. The *basic block generator* divides bytecode instructions into a collection of basic blocks. The *def* instruction and *use* instruction 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 *definition and use information generator* generates a DUI by using the method's CDFG.

The following section explains how the DUIs are used to increase testability, controllability, and observability for an OO component.

4. INCREASING OO COMPONENT TESTABILITY

Testing software is easier when testability is high, and, in general, increasing testability makes it easier to detect faults. This section explains the process used to increase OO component testability. The DUIs from the previous step are used to increase testability by influencing the two factors of *controllability* and *observability*.

4.1. Increasing controllability

Binder [7] defined the significance of controllability as 'if users cannot control the inputs, they cannot be sure what caused a given output.' Controllability focuses on the ease of controlling a component's inputs. This means that an OO component that supports various ways of supplying inputs to exercise the component tends to provide better controllability.

To increase controllability, the DUIs mentioned in the previous step are used to collect definition–use pairs of variable between last-definitions and first-uses. The definition–use pairs of a variable are called *definition–use couplings for testing* (*DUCoT*). For example, for the variable *total* of the vending machine in Table III, 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 position y as the *last-def location* x and the *first-use location* y. 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 lastdefinitions 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.

Figure 8 shows the DUCoTs for variables of vending machine. From the figure, the first DUCoT is for variable curQtr, DUCoT(curQtr) $= (D_L, U_F)$. D_L is a set of four elements, ⟨*init*⟩[11], *addQtr*[7], *returnQtr*[2], and *vend*[109]. Each element contains two parts, M_d and $[L_d]$. The first part, M_d, is a method's name. The second part, L_d, is a last-def location in method M_d. For example, the last element of D_L, vend[109], indicates a method *vend* and a last-def location 109. U_F is a set of two elements, *addQtr*[2] and *vend*[11]. Each element contains two parts, M_u and $[L_u]$. The first part, M_u, is a method's name. The second part, L_u, is a first-use location in method M_u. For example, the last element of U_F, vend[11], indicates a method *vend* and a first-use location 11.

```
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([5]} )

DUCoT (availType)   =  ( {<init>[21]}, {available[1]} )
```

Figure 8. The DUCoTs of vending machine's variables.

The DUCoTs are used to increase controllability by supporting test case generation to cover all the necessary tests. These test cases are generated according to the CBT criteria proposed by Jin and Offutt [16], as described in Section 2.3. The coupling-based testing criteria are a collection of rules that impose requirements on a set of test cases. Applying CBT to component testing requires some minor modifications to the terminology.

The *ACU* 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 *ACU* for a component by using DUCoT is given in the following definition.

Definition: Let (D_L, U_F) be a DUCoT of variable v. The *ACU* of variable v is defined as

$$\mathbf{AllCoU(v) = ((D_L \times U_F) = \{M_d[L_d], M_u[L_u]) | \forall M_d[L_d] \in D_L \text{ and } \forall M_u[L_u] \in U_F)}$$
$$\mathbf{\text{and } M_d = M_u \rightarrow L_d > L_u\}}$$

This paper describes *test requirements* as definition–use pairs that are to be tested. The test requirement for an ordered definition–use 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 *ACU* considers definition–use pair between methods, a last-def in a method and a first-use in another method. Therefore, if the first-use and last-def locations are in the same method, the first-use location must appear before the last-def location. Examples of the test requirements of ACU of vending machine are shown in Table IV.

A tool to automatically generate DUCoTs and test requirements for an OO component was implemented. The tool is composed of two main modules; a module to find first-uses and last-defs of variables and a module to create the ACU test requirements as shown in Figure 9. The *first-use and last-def generator* uses DUIs to generate DUCoTs. The *coupling-based test generator* uses DUCoTs to generate test requirements for ACU.

According to the test requirements generated, test cases are then created to satisfy these requirements. A test case is a sequence of method calls. For example, one test requirement for the du-pair $(M_d[L_d], M_u[L_u])$ causes the execution to reach two specific locations. The first is the last-def location L_d of method M_d, which defines that variable (*required def location*). The second is the first-use location L_u of method M_u, which uses that same variable *required use location*. Moreover,

Table IV. The test requirements of all-coupling-uses of vending machine.

No.	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 [5]
13		vend [7], available [5]
14	availType	⟨init⟩ [21], available [1]

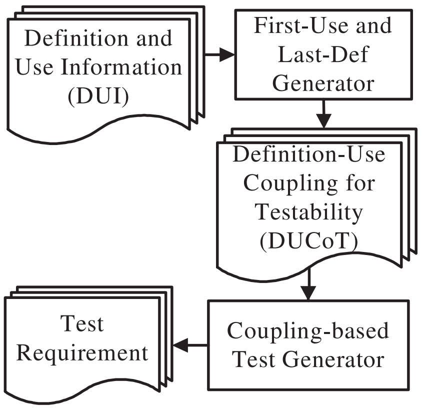

Definition and Use Information (DUI)
First-Use and Last-Def Generator
Definition-Use Coupling for Testability (DUCoT)
Coupling-based Test Generator
Test Requirement

Figure 9. Test requirement generation process.

this execution must not redefine the variable between these two locations. That is, this must be a *def-clear path execution*. Some test requirements are infeasible because the required locations cannot be reached or the execution does not produce a def-clear path. Instrumentation is used to eliminate infeasible paths during test case generation.

The overall process of test case generation is shown in Figure 10. The solid boxes represent steps that are done automatically and the dashed boxes represent steps that are done manually. The *definition-use track instrumentation* was implemented to automatically instrument Java .class files and to create the *instrumented class*. The instrumentation is performed at the bytecode level by inserting auxiliary instructions at all definition and use locations. The definition and use locations are indicated by DUIs defined in Section 3.2. The inserted instructions are used to record the

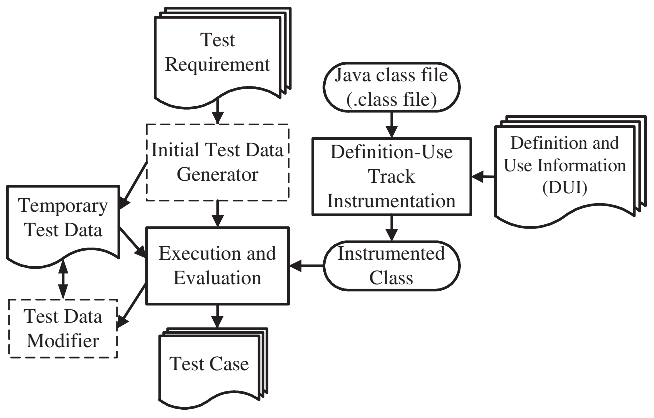

Test Requirement
Java class file (.class file)
Initial Test Data Generator
Definition-Use Track Instrumentation
Definition and Use Information (DUI)
Temporary Test Data
Execution and Evaluation
Instrumented Class
Test Data Modifier
Test Case

Figure 10. Test case generation process.

reached locations. The *temporary test data* are first generated by the *Initial Test Data Generator* module according to a test requirement. The *execution and evaluation* executes the *instrumented class* against the *temporary test data* and records the sequence of reached locations.

The *execution and evaluation* module also automatically checks the required def and use locations and checks whether the execution is a def-clear path execution by using the sequence of reached locations. If the execution is not a def-clear path or the required locations are not reached, the *test data modifier* applies a simple search process. It modifies the *temporary test data* by adding a method call or by changing the values of the method call parameters. The initialization and modification of the test data are done manually. For example, in Table IV, the fourth test requirement, (addQtr[7], vend[11]), requires the execution to reach location 7 of method *addQtr* and location 11 of method *vend*.

The initial test data for this test requirement consist of the vending machine class object, the call to method *addQtr*, and the call to method *vend*. The parameter of method *vend* is randomly generated, e.g. 1. These initial test data—*new()*, *addQtr()*, *vend(1)*—when executed, reach location 7 of method *addQtr* and location 11 of method *vend* and follow a def-clear path execution. Therefore, the test case for this test requirement is *new()*, *addQtr()*, *vend(1)*. From the same table, consider the ninth test requirement, (⟨init⟩[6], vend[94]), which requires the execution to reach location 6 of method ⟨*init*⟩ and location 94 of method *vend*. The initial call, ⟨init⟩, is called automatically by the system when a new class object, new(), is created. Hence the test data are initialized, using the same parameter as before, to be *new()*, *vend(1)*. When that test is executed, location 94 of method *vend* could not be reached. To correct this, the test data are modified by adding calls to method *addQtr* and changing the parameter value of method *vend* to 2. As a result, the modified test case of the ninth test requirement in Table IV is *new()*, *addQtr()*, *addQtr()*, and *vend(2)*. Automatically initializing and modifying test data are complex problems, and left as future work.

The step-by-step test case generation process for a test requirement for a variable with parameters, location L_d of method M_d and location L_u of method M_u ($M_d[L_d]$, $M_u[L_u]$), is described as follows:

1. First, initial test data are generated by creating a new object, then making a method call to M_d and later a method call to M_u. Note that the initial test data, specifically the method parameters, are randomly generated to fulfil requirements. These data are called 'temporary test data.'
2. Next, the instrumented class is executed with the temporary test data. The sequence of locations reached is recorded during execution.
3. After that, the locations reached are evaluated. 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 the variable between location L_d and location L_u, then the temporary test case is finalized and established as the 'test case.' Otherwise, the temporary test data are modified by adding a method call or by changing the values of the method call arguments, and step 2 is repeated.

The result of test case generation for a test requirement is the sequence of creating a new object and making method calls that cause execution of the required def location and the required use location with a def-clear path.

4.2. Increasing observability

Binder [7] makes the following point about observability in OO 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 test engineers to determine whether the software behaves correctly during testing. The observability of black-box software is inherently limited because only the outputs are visible. The encapsulation and data-hiding properties of black-box software cause problems with testing by making OO software less observable [6]. This is because the internal state is not readily available. This makes it possible for the internal state to be erroneous while still yielding correct outputs. Hence, being able to access the internal state is a crucial task for testing.

An approach to access the internal state is based on debugging. A debugger generally provides a view of *all* the state information at a certain point during an execution. The larger a program is, the less observable it is. To alleviate this problem, we derived a method for specifying particular observation points to observe internal states of variables during testing. Typically, internal states of classes are not immediately and directly available to test engineers. For example, from Figure 4, the state variable *total*, as defined in line 31, is a private variable; hence, the client could not access it directly. If line 31 has a fault, no test case could detect it. To deal with this problem, being able to observe intermediate values of state variables during testing by using temporary variables is necessary.

This paper introduces a technique called *observability probes*. Observability probes keep track of relevant internal state variables at their definition and use locations and assert the correctness of such state variables during testing. Such instrumentation is used to help the observability probes process.

The intermediate values of state variables can be observed by inserting auxiliary instructions at the bytecode level. Such instructions are inserted into OO components in such a way that the overall program behaviour is not changed (of course, these instruments could affect the timing of a real-time process, but this research explicitly does not address real-time software). The instrumented instructions are classified into three groups. The first defines instructions for creating temporary variables called *tracking variabl*es for each variable in the component. These variables are defined to be public; hence, they can be accessed by other test classes. For example, variables *d_curQtr* and *u_curQtr* are created to be tracking variables of variable *curQtr* for storing its values at its definition location and its use location. The second group defines instructions for storing the values of each variable into its corresponding tracking variable. For example, the variable *d_curQtr* is assigned the value of the variable *curQtr* at its definition location. Similarly, the variable *u_curQtr* is assigned the value of the variable *curQtr* at its use location. The second group of instructions is inserted at every location of the variable definition and every location of the variable use.

The last group contains instructions to assert the correctness of the values of the internal state variables. The assertion is performed at every use location. The assertion is to see whether the temporarily stored value at the use location is equal to the temporarily stored value of the

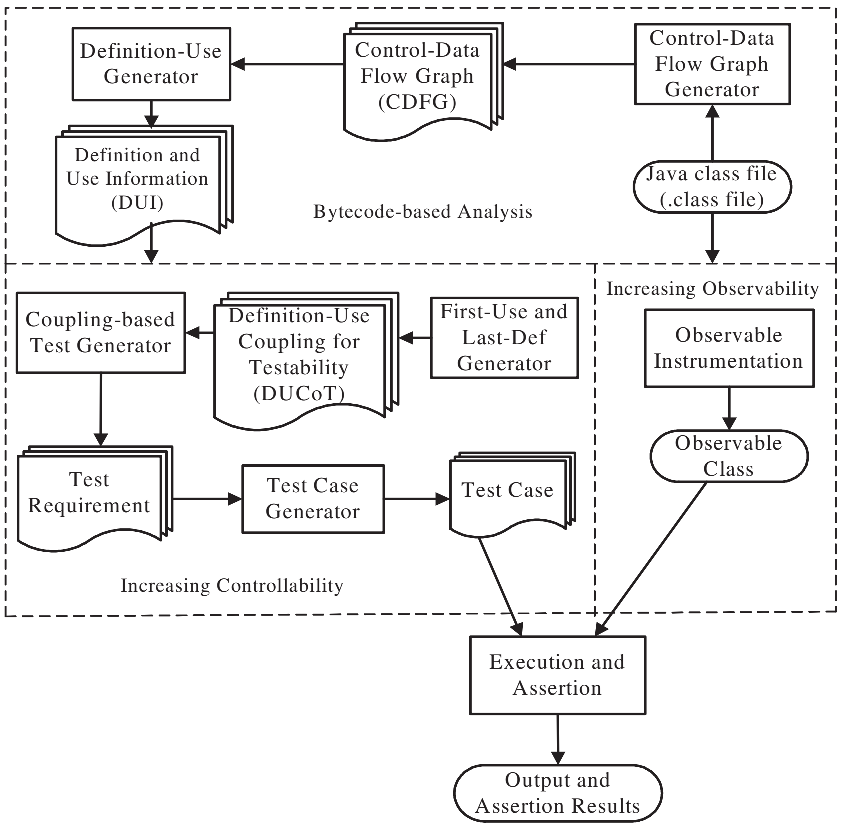

Definition-Use Generator
Control-Data Flow Graph (CDFG)
Control-Data Flow Graph Generator
Definition and Use Information (DUI)
Bytecode-based Analysis
Java class file (.class file)
Increasing Observability
Coupling-based Test Generator
Definition-Use Coupling for Testability (DUCoT)
First-Use and Last-Def Generator
Observable Instrumentation
Observable Class
Test Requirement
Test Case Generator
Test Case
Increasing Controllability
Execution and Assertion
Output and Assertion Results

Figure 11. The process of increasing object-oriented component testability.

corresponding state variable at the definition location, i.e. $u\_curQtr = d\_curQtr$. If the two values associated with the state variable are equal, that variable retains its value (a def-clear path was followed). Otherwise, the variable has been unexpectedly redefined. Refer to Table IV, which provides all test requirements of all relevant state variables for the vending machine class example.

For example, all test requirements defined by ordered pairs from lines 1 to 8 must be executed to test the variable *curQtr*. Specifically, the first test requirement (⟨*init*⟩[11], addQtr[2]) is translated to the test case *new()*, *addQtr()*, as depicted in Figure 10. The *new()* method makes a call to the method ⟨*init*⟩. When this test case is executed, the definition location (line 11 in Figure 5) of the method ⟨*init*⟩ and the use location (line 2 in Figure 5) of the method *addQtr* have been reached. At the point of the definition and use locations, the state variables of variable *curQtr* are stored into their tracking variables, *d_curQtr* and *u_curQtr*. At the point of the use location, an assertion is executed to compare the value of the tracking variable at its use location (*u_curQtr*) with that at its definition location (*d_curQtr*).

This observability probe process has been implemented. A Java .class file under test is instrumented by the *Observable Instrumentation* module to add the three groups of instructions. The *Observable Class* is produced by referring to the definition and use locations of the DUIs, described in Section 3.2.

Figure 11 shows the overall process of increasing OO component testability. From the figure, an OO component (.class file) is analysed at the bytecode level by the *betycode-based analysis* process. This produces the DUI for each method of the OO component under test. The DUIs are then used by the *increasing controllability* process to generate test cases and used by the *increasing observability* process to monitor internal state variables. The *increasing controllability* process generates test cases according to the ACU criterion for the OO component under test. The *increasing observability* process instruments an OO component to produce the *observable class*. When the *observable class* is executed by the *execution and assertion* module against test cases, the execution results include the normal program outputs and the internal state variable assertion.

5. CASE STUDY

This section demonstrates the effectiveness of the process for increasing OO component testability. As discussed in Section 4, the OO component testability is increased by increasing controllability and observability. To increase controllability, we support ways to generate test cases based on *ACU*. To increase observability, observability probes are added to trace and assert internal state variables. The increased OO component testability makes it easier to recognize failures.

Increasing the testability of OO components should make it easier to detect faults. Mutation analysis is used to seed faults into OO components and thereby evaluate whether increasing the testability makes it easier to detect the faults. 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. Andrews *et al.* [21] 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 [22], which found that tests that detect mutation-like faults are good at detecting more complicated faults.

As explained in the background section, mutation analysis works by modifying copies of the original program or component. *Mutants* are created by making copies of the original version, then

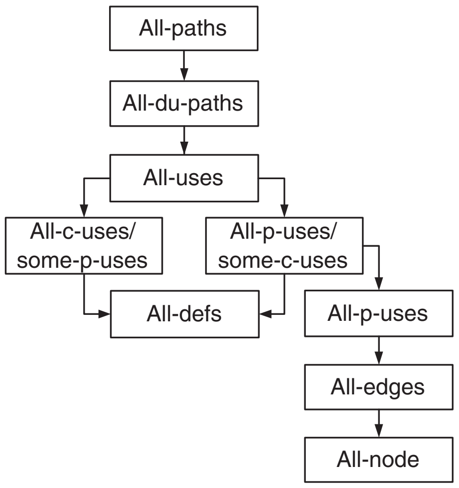


Figure 12. A family of data flow testing criteria.

inducing each mutant change into a unique copy. Tests are run on the original version, and then each mutant version. If the results from an original version are different from the results from a mutant version, the mutant is said to be *killed* by the tests.

Figure 12 shows a subsumption relationship in a family of data flow testing criteria by Rapps and Weyuker [14]. The relationships show that all-du-paths (ADUP) subsumes AU. AU requires a test to cover a path from each definition to reachable use, whereas ADUP requires tests to cover all possible definition–use paths.

This paper generates test cases based on the ACU criterion as explained in Section 4.1. This criterion requires, for each coupling-def, that at least one test case executes a path from the definition to each reachable coupling-use. Both ACU and AU consider the paths from each definition to reachable uses. The ACU criterion considers only the definition–use paths of variables between the last-definition in a method and the first-uses in other methods. Consequently, the definition–use pairs of the ACU criterion form a subset of the definition–use pairs of the AU criterion. The intra-procedural ADUP criterion subsumes the intra-procedural AU criterion. Although intra-procedural criteria cannot be directly compared with inter-procedural criteria, it seems reasonable to think of ADUP as being stronger than the inter-procedural version of AU, ACU. To make a stronger case for the value of the observability probes, we compare tests generated from the ACU criterion with observability probes with tests generated from the ADUP criterion. This introduces a modest bias towards ADUP. The design of the experiment is described in the following subsection.

5.1. Experimental setting and results

The experiment proceeded in six steps:

1. Prepare classes to test.
2. Generate a set of test cases following the approach, *all-coupling-uses*, for each class.

3. Generate a set of test cases to satisfy *ADUP* coverage for each class.
4. Generate the mutants for each class.
5. Run each set of test cases on the original and each mutant version.
6. Compute the fault detection ability of each set of test cases.

The study used five subjects, the vending machine class (VendingMachine) and four classes from a data structure package; StackArr, QueueArr, BinaryHeap, and ArrList. Table V shows statistics for each class. Column 'SLOC' shows the source lines of code, '#Instructions' shows the number of bytecode instructions, '#Variables' shows the number of variables, and '#Methods' shows the number of methods. Following the method explained in Section 4.1, test cases were generated to satisfy the ACU criterion for each class. The observations of internal states were added by observability probes as explained in Section 4.2. These are called *ACU with observability* tests (*ACU-O tests*). Also, a set of test cases for each class was generated to satisfy the *ADUP* criterion (*AllDU* tests). Table VI shows the number of ACU-O and AllDU tests for each class. The obvious observation is that in all classes AllDU require more tests than ACU-O. This is consistent with the subsumption relationship [14] that if a more stringent criterion is chosen, the number of required definition–use pairs increases.

This work is interested in faults in the data state, namely, *incorrect data state*. Incorrect data states are classified into two types. The first type, *incorrect definition*, occurs when a variable value is defined incorrectly. The second type, *incorrect use*, occurs when a variable value is used incorrectly. The incorrect data state is generated by injecting a fault into the data state. This is called a *data state mutation*. This generation process is similar to the interface mutation process [17]. On the basis of the incorrect data state, we define two data state mutation operators: *definition replacement* and *use replacement*. The definition replacement operator creates a mutant by redefining the variable's value. The use replacement operator creates a mutant by changing the value used to an incorrect value. Data state mutation creates mutants by applying data state mutation operators at definition

Table V. Description of each class used in the experiment.

Class name	SLOC	#Instructions	#Variables	#Methods
VendingMachine	40	118	4	5
StackArr	58	140	3	9
QueueArr	59	151	5	8
BinaryHeap	61	201	3	8
ArrList	58	229	4	9

Table VI. The number of ACU and AllDU tests.

Class name	ACU	AllDU
VendingMachine	13	22
StackArr	25	25
QueueArr	22	27
BinaryHeap	22	39
ArrList	58	68

and use locations at the bytecode level. The definition replacement operator is applied at definition locations and the use replacement operator is applied at use locations. For example, at the definition location l, the variable x is mutated to generate the mutant by redefining the value of variable x.

The experiment assumes that the five subject classes are fault-free. The discovered faults are only from mutants. The number of mutants generated for each class is shown in Table VII. Each original class and its mutants were executed against all ACU-O and AllDU tests. A mutant is said to be killed by a set of test cases if the execution results of the mutant are different from those results of the corresponding original class. For example, if the results from executing the ACU-O tests on the VendingMachine class and a mutant are different, the mutant is killed by the ACU-O tests. When a mutant is killed, it means a fault is detected. The fault detection scores were computed in terms of the number of faults detected for each class.

Table VIII shows the fault detection ability of the ACU-O and AllDU tests. Column '#Mutants' indicates the number of mutants for each class, '#Tests' indicates the number of test cases for each class, 'Faults Detected' indicates the number of mutants killed for each class, and 'Detection Increase' indicates the number of additional mutants that were killed by the ACU-O tests.

The execution results provided by this experiment indicate that the fault detection ability of ACU-O tests was 73% greater than that of the AllDU tests for all classes. The increased fault detection ability of ACU-O tests was mostly affected by the observability probes. The observability probes help observe the validity of the internal state variables during testing. For example, consider the VendingMachine class's mutant that changed the *total* value used at line 31 in Figure 4. This fault was not directly propagated to the output and this mutant was not killed by the AllDU tests, even though the correct du-path was executed. By using the observability probes, the ACU-O tests were able to detect the incorrect *total* value. As shown in Table VIII, the ACU-O tests detect all

Table VII. The number of mutants for each class.

Class name	#Mutants
VendingMachine	18
StackArr	13
QueueArr	21
BinaryHeap	25
ArrList	36

Table VIII. Case study results on all-du-paths and all-coupling-uses with observability.

		AllDU			ACU-O				
Class name	#Mutants	#Tests	Faults detected	%Faults detected	#Tests	Faults detected	%Faults detected	Detection increase	%Detection increase
VendingMachine	18	22	13	72	13	18	100	5	39
StackArr	13	25	8	62	25	13	100	5	63
QueueArr	21	27	15	71	22	21	100	6	40
BinaryHeap	25	39	6	24	22	25	100	19	317
ArrList	36	68	21	58	58	32	89	11	25
Sum	113	181	63		140	109		46	

the faults for four out of five classes. The fault detection ability of ArrList class was not 100% because of the *dd* data flow anomaly that occurred when a definition is followed by another identical definition without any intermediate use. For example, from the ArrList class, the partial instructions shown in Figure 13 created the *dd* data flow anomaly for the variable *end*. As part of the future work, data flow anomalies should be eliminated before testing.

Our experiment can observe that applying our all-coupling-uses criterion (ACU-O tests) to generate test cases satisfied more definition–use pairs than the ADUP criterion (AllDU tests). As shown in Table IX, the def-clear path executions of ACU tests reached 92% of definition–use pairs, whereas the def-clear path executions of AllDU tests reached only 74%.

As a conclusion, the increasing OO component testability process in this paper provides an improvement in the ability for testers to detect faults. The process helps generate effective tests for OO components

```
...
31:  aload_0
32:  aload_0
33:  dup
34:  getfield       ArrList.end I (4)
37:  iconst_1
38:  iadd
39:  dup_x1
40:  putfield       ArrList.end I (4)        // definition end
43:  aload_0
44:  getfield       ArrList.array [I (2)
47:  arraylength
48:  irem
49:  dup_x1
50:  putfield       ArrList.end I (4)        // definition end
...
```

Figure 13. The *dd* data flow anomaly of variable *end* ArrList class.

Table IX. The number of definition–use pairs and the number of def-clear path executions.

	AllDU		ACU	
Class name	#Definition–use pairs	#Def-clear path executions	#Definition–use pairs	#Def-clear path executions
VendingMachine	27	22	14	13
StackArr	31	25	31	25
QueueArr	29	27	23	22
BinaryHeap	73	39	24	22
ArrList	86	68	61	58
Sum	246	181	153	140

6. CONCLUSIONS AND FUTURE WORK

This paper provided an analysis method on bytecode instructions to automatically gather control flow and data flow information. This information is represented by an intermediate graph, CDFG, for each method in an OO component. The CDFGs are used with the ACU criterion to increase OO component testability by supplying test cases to exercise the component as necessary (increasing controllability) and making it easier to observe internal state variables during testing (increasing observability). The increasing process helps generate effective tests for OO components and provides an improvement in the fault detection ability. Further details are available in Kansomkeat's dissertation [23]. From the case study, the fault detection ability of the test cases generated from the ACU criterion with observability probes is greater than that of test cases generated from ADUP criterion.

The method described in this paper is not completely automated. As shown in Figure 10 in Section 4.1, the initialization and modification of the test data must be done manually. Future efforts will focus on improving the tools to automate all processes. In the near future, state variables at the control flow will be tracked along with a bytecode-based analysis process to help the automatically initialization process. We will also investigate new technique and adaptation to existing dynamic techniques such as search-based techniques that are required to test data modification processes This paper focuses on intra-class method calls, but extending the inter-class method calls is straightforward. However, considering multiple classes is more complex when class hierarchies with dynamic-type binding and polymorphism are used. Instances of class hierarchies are achieved by allowing state variables to be references to some other object. A virtual class is needed to mange the complexity. One instance of this class can access class variables of methods in the separate class. Therefore, the definition and use information explained in Section 3.1 can be gathered from the virtual class. We hope to explore these ideas in future work

ACKNOWLEDGEMENTS

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.

REFERENCES

1. Beizer B. *Software Testing Techniques* (2nd edn). Van Nostrand Reinhold, Inc.: New York, NY, 1990. ISBN 0-442-20672-0.
2. Abdurazik A, Offutt J. Generating test cases from UML specifications. *Second International Conference on the Unified Modeling Language* (*UML '99*), Fort Collins, CO, October 1999; 416–429.
3. Gallagher L, Offutt J, Cincotta A. Integration testing of object-oriented components using finite state machines. *Software Testing*, *Verification and Reliability* 2007; **17**(1):215–266.
4. Hong HS, Kwon YR, Cha SD. Testing of object-oriented programs based on finite state machines. *The Asia-Pacific Software Engineering Conference* (*ASPEC95*), Brisbane, Australia, December 1995; 234–241.
5. IEEE Standard Glossary of Software Engineering Technology. *ANSI/IEEE 610.12*. IEEE Press: New York, 1990.
6. Voas JM, Miller KW. Software testability: The new verification. *IEEE Software* 1995; **12**(3):17–28.
7. Binder RV. Design for testability with object-oriented systems. *Communications of the ACM* 1994; **37**(9):87–101.
8. Freedman R. Testability of software components. *IEEE Transactions on Software Engineering* 1991; **17**(6): 553–563.

9. Wang Y, King G, Fayad M, Patel D, Court I, Staples G, Ross M. On built-in test reuse in object-oriented framework design. *ACM Journal on Computing Surveys* 2000; **32**(1):7–12.
10. Weyuker E. Testing component-based software: A cautionary tale. *IEEE Software* 1998; **15**(5):54–59.
11. Kansomkeat S, Offutt J, Rivepiboon W. Bytecode-based analysis for increasing class-component testability. *ECTI-CIT Transactions on Computer and Information Technology* 2006; **2**(2):33–44.
12. Lindholm T, Yellin F. *The JavaTM Virtual Machine Specification* (2nd edn). Addison-Wesley: Reading, MA, 1999.
13. Laski J, Korel B. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering* 1983; **9**(3):347–354.
14. Rapps S, Weyuker EJ. Selecting software test data using data flow information. *IEEE Transaction on Software Engineering* 1985; **11**(4):367–375.
15. Harrold MJ, Rothermel G. Performing data flow testing on classes. *The ACM SIGSOFT Foundation of Software Engineering*, New Orleans, LA, 1994; 154–163.
16. Jin Z, Offutt J. Coupling-based criteria for integration testing. *Software Testing*, *Verification and Reliability* 1998; **8**(3):133–154.
17. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practical programmer. *IEEE Computer* 1978; **11**(4):34–41.
18. DeMillo RA, Offutt J. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 1999; **17**(9):900–910.
19. Aho AV, Sethi R, Ullman JD. *Compilers*: *Principles*, *Techniques and Tools*. Addison-Wesley: Reading, MA, 1986.
20. Apache Software Foundation. *BCEL*: *Byte Code Engineering Library*. Part of the Apache/Jakarta project, 2002–2003. http://jakarta.apache.org/bcel/ (accessed July 2007).
21. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? *The 27th International Conference on Software Engineering*, St. Louis MI, U.S.A., 2005; 402–411.
22. Offutt J. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology* 1992; **1**(1):3–18.
23. Kansomkeat S. An analysis technique to increase testability of class-component. *Dissertation*, Chulalongkorn University, 2006; 1–67.