Roberto Paulo Andrioli de Araujo

Scalable data-flow testing

São Paulo

2014

Roberto Paulo Andrioli de Araujo

Scalable data-flow testing

Submitted to the Graduate Program in Information Systems of the School of Arts, Sciences and Humanities – University of São Paulo – in partial fulfillment of the requirements for the degree of Master of Science.

Corrected version. The original version is found in the Library of School of Arts, Sciences and Humanities and in the Digital Library of Theses and Dissertations of University of São Paulo.

Supervisor: Marcos Lordello Chaim

São Paulo

2014

CATALOGING IN PUBLICATION (CIP) DATA
(University of Sao Paulo. School of Arts, Sciences and Humanities. Library)

Araujo, Roberto Paulo Andrioli de
Scalable data-flow testing / Roberto Paulo Andrioli de Araujo ; supervisor, Marcos Lordello Chaim. – Sao Paulo, 2014
100 p. : ill.

Dissertation (Master of Science) - Graduate Program in Information Systems, School of Arts, Sciences and Humanities, University of Sao Paulo, Sao Paulo, 2014
Corrected version

1. Software testing and evaluation. 2. Software verification and validation. 3. Software engineering. 4. Algorithms and data Structures. I. Chaim, Marcos Lordello, sup. II. Title.

CDD 22.ed. – 005.14

This dissertation was defended on

September 15, 2014

and approved by

Marcos Lordello Chaim
University of São Paulo

Plínio Roberto Souza Vilela
CFlex

Sudipto Ghosh
Colorado State University

To Darcy, Elzearia and Evandro.

Abstract

ARAUJO, Roberto Paulo Andrioli de. **Scalable data-flow testing**. 2014. 100 p. Dissertation (Master of Science) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, 2014.

Data-flow (DF) testing was introduced more than thirty years ago aiming at verifying a program by extensively exploring its structure. It requires tests that traverse paths in which the assignment of a value to a variable (a definition) and its subsequent reference (a use) is verified. This relationship is called definition-use association (dua). While control-flow (CF) testing tools have being able to tackle systems composed of large and long running programs, DF testing tools have failed to do so. This situation is in part due to the costs associated with tracking duas at run-time. Recently, an algorithm, called Bitwise Algorithm (BA), which uses bit vectors and bitwise operations for tracking intra-procedural duas at run-time, was proposed. This research presents the implementation of BA for programs compiled into Java bytecodes. Previous DF approaches were able to deal with small to medium size programs with high penalties in terms of execution and memory. Our experimental results show that by using BA we are able to tackle large systems with more than 250 KLOCs and 300K required duas. Furthermore, for several programs the execution penalty was comparable with that imposed by a popular CF testing tool.

Keywords: Data-flow testing. Program instrumentation. Coverage analysis at run-time.

Resumo

ARAUJO, Roberto Paulo Andrioli de. **Scalable data-flow testing**. 2014. 100 f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2014.

Teste de fluxo de dados (TFD) foi introduzido há mais de trinta anos com o objetivo de criar uma avaliação mais abrangente da estrutura dos programas. TFD exige testes que percorrem caminhos nos quais a atribuição de valor a uma variável (definição) e a subsequente referência a esse valor (uso) são verificados. Essa relação é denominada associação definição-uso. Enquanto as ferramentas de teste de fluxo de controle são capazes de lidar com sistemas compostos de programas grandes e que executam durante bastante tempo, as ferramentas de TFD não têm obtido o mesmo sucesso. Esta situação é, em parte, devida aos custos associados ao rastreamento de associações definição-uso em tempo de execução. Recentemente, foi proposto um algoritmo — chamado *Bitwise-Algorithm* (BA) — que usa vetores de bits e operações bit a bit para monitorar associações definição-uso em tempo de execução. Esta pesquisa apresenta a implementação de BA para programas compilados em Java. Abordagens anteriores são capazes de lidar com programas pequenos e de médio porte com altas penalidades em termos de execução e memória. Os resultados experimentais mostram que, usando BA, é possível utilizar TFD para verificar sistemas com mais de 250 mil linhas de código e 300 mil associações definição-uso. Além disso, para vários programas, a penalidade de execução imposta por BA é comparável àquela imposta por uma popular ferramenta de teste de fluxo de controle.

Palavras-chave: Teste de fluxo de dados. Instrumentação de programas. Análise de cobertura em tempo de execução.

List of Figures

List of Tables

Contents

1 Introduction

The purpose of software testing is to detect anomalies in program execution, and in failing to detect them, to increase one's confidence in the program correctness. These anomalies are usually caused by defects that, for various reasons, have gone unnoticed during the development phase [1]. These anomalies are called *failures*.

To detect failures, test cases are developed and executed on the program. A test case aims at producing an output that is different from the expected. Test cases with such a property are successful tests (i.e., they detect the presence of a fault in a program) [2].

Software that comes with successful test cases increases our confidence in its quality. If a change in the program code causes a failure, the existing test set or newly created test cases should detect it. If the test set execution is automated, test cases can be executed on every change in the code, allowing the detection of bugs before the release of the software. To achieve this level of confidence test cases should be designed rigorously and systematically.

Testing techniques define the so-called *testing entities*. Testing entities are the program *requirements* that should be exercised by a test set. There exist different techniques that establish testing requirements. Two of the best known are *structural* and *functional* testing techniques.

Functional testing attempts to assure that the program specifications are verified; thus, the requirements of the functional testing are the features described in the specification. This kind of testing is independent of the program code and depends only on the specification. On the other hand, structural testing requires the analysis of the program source or object code to determine the requirements that must be exercised by tests. This is why the former is called *black-box* testing — it verifies the program from its features — and structural testing is called *white-box* testing — it verifies the program from its structure.

Structural testing requirements can be divided into two groups — *control-* and *data-flow* testing. In the first one, the testing requirements are derived from the flow graph obtained from a program. Examples of control-flow testing requirements are: *node*, *edge* and *path*, where a node is a set of statements executed in sequence, an edge is the possible transfer of control between nodes, and a path corresponds to a sequence of nodes.

Data-flow (DF) testing, in turn, involves the development of tests which exercise every value assigned to a variable and its subsequent references (uses) occurring either in a computation or in a predicate. These requirements are called *definition-use associations*

(dua). The underlying intuition of DF testing is that by testing the values and their uses one would increase the confidence in the program correctness [3, 4].

Code coverage involves determining the structural testing requirements covered by a test set. It is a measure of the quality of the test set with respect to a testing criterion. Examples of code coverage are *statement coverage*, *edge coverage* and *all-uses coverage* [3, 4].

This work is targeted at improving DF testing to make it scalable to programs developed at industrial settings.

1.1 Motivation

An important part of current software industry, known as internet-based companies, works in perpetual development mode in which new features are made available to users on a daily basis. As a result, the systems are continuously growing and being deployed. Feitelson et al. [5] describe Facebook development and deployment process in which new code is released at high rate, several times a working day.

To develop quality software at such a fast rate, these companies rely in part on extensive automated testing [6]. Browser-based automated testing and *xUnit* frameworks are used to execute thousands of tests. In addition, tools automatically collect code coverage data to assess the code produced by engineers. In this context of continuous deployment, coverage tools are useful only if they are able to produce coverage data equally fast.

To automatically determine the testing requirements covered by a test set, the program (object or source) code should be instrumented. Program instrumentation inserts additional code in a program to collect coverage information during its execution. Instrumentation makes programs run slower and causes greater memory consumption. Furthermore, inefficient program instrumentation might render effective software testing techniques impractical for industrial use.

Tools that support control-flow (CF) testing have widespread use in industry. Although the tools performance may vary depending on the characteristics of the systems, commercial (e.g., Clover[1]) and open-source (e.g., Cobertura[2], JaCoCo[3]) tools have been utilized to assess control coverage of large systems. Moir reports that JaCoCo was run against 37,000 Eclipse JDT core tests with an execution overhead of 2% [7].

On the other hand, one hardly finds DF testing in use at industrial settings, even after more than thirty years its introduction and despite having been shown to be an effective testing technique [8, 9], adequate for security assessment [10], and useful to

[1] <http://www.atlassian.com/software/clover/>
[2] <http://cobertura.github.io/cobertura/>
[3] <http://www.eclemma.org/jacoco/>

support fault localization [11]. In part, this situation can be explained by the fact that DF supporting tools are not scalable for large systems due to the costs associated with tracking duas at run-time.

Other factor that increases the cost of DF testing is that it takes into account both control- and data-flow information to derive requirements to be tested. As a result, more requirements need to be exercised by a test set in comparison to CF only testing. Thus, more effort is required to fulfill a DF criterion.

Some DF testing instrumentation techniques [12, 13] were proposed to address the costs associated with monitoring duas at run-time. However, these approaches rely on complex computations and expensive data structures to collect dua coverage. Recently, Chaim and Araujo [14] proposed a novel algorithm, called Bitwise Algorithm (BA), to tackle this issue. The new algorithm utilizes efficient bitwise operations and inexpensive data structures to track intra-procedural duas (i.e., duas occurring within a procedure). Simulations show that BA is at least as good as the most efficient DF instrumentation techniques and that it can be up to 100% more efficient [14].

1.2 Objectives

The main goal of this research project is to develop a tool that supports intra-procedural DF testing of programs compiled into Java bytecodes using the BA approach to track duas at run-time. To achieve such a goal, the BA-DUA — Bitwise Algorithm-powered Definition-Use Association coverage — tool was developed.

Another goal is to assess the scalability of BA-DUA with respect to software developed in industry. Our evaluation takes into account two types of penalties imposed by instrumenting programs with BA: execution overhead, given by the extra time needed to execute the instrumentation code with respect to an uninstrumented program; and memory overhead, represented by the growth of the object code due to the insertion of BA code.

Thus, the following specific objectives for this project were defined:

- to identify the main strategies and tools developed to support DF testing;
- to develop a scalable open-source DF testing tool for programs compiled into bytecodes based on the BA approach;
- to assess the scalability of the new tool in applications comparable to those developed in the industry;
- to propose improvements to the original BA.

1.3 Key findings

We assessed the penalties imposed by BA-DUA in ten programs ranging from 843 to 267,707 lines of code (LOC) with a total number of duas ranging from 1,186 to 315,092. The test set of these programs were run both, without any instrumentation, and with BA-DUA and JaCoCo instrumentation.

Our results indicate that the BA execution overhead varied from negligible to 125%, with an average of 28%. The BA-DUA overhead was over 25% for only two subjects in ten programs. Furthermore, for several programs the BA-DUA execution penalty was comparable with that imposed by JaCoCo. In terms of program growth, the BA memory overhead ranged from 32% to 107%, with the average being 57%.

Previous approaches for dua monitoring were able to deal only with small to medium size programs with sizeable penalties. By using BA-DUA we are able to tackle large systems with more than 250 KLOC and 300K required duas with fairly moderate overhead. Our experimental data suggests that BA allows intra-procedural DF testing to be used in a much bigger class of systems.

1.4 Organization

In this chapter, we presented the context, motivation and objectives of our research whose goal is to develop and assess a tool to support scalable DF testing. The rest of this Master's thesis is organized as follows:

- Chapter 2 presents the fundamental concepts in software testing;
- Chapter 3 examines related work;
- Chapter 4 describes the Bitwise Algorithm (BA) for tracking intra-procedural DF testing requirements;
- Chapter 5 describes the Java Virtual Machine and the DF analysis in its context;
- Chapter 6 shows the implementation of BA-DUA and the instrumentation of programs compiled into bytecodes;
- Chapter 7 contains the evaluations of the BA-DUA;
- Finally, Chapter 8 contains the conclusions.

2 Fundamental testing concepts

In this chapter, we present the fundamental concepts of software testing and the different testing techniques. In particular, the concepts related to structural testing — the focus of our research — are detailed.

2.1 Testing techniques and criteria

Testing techniques are used to assess various aspects of the program. Different techniques attempt to reveal the presence of failures in a program execution. In doing so, they augment one's confidence that the features of a program are correct.

Testing techniques define the so-called *testing requirements*. They characterize what should be exercised by a test set and can be obtained by analyzing the specification or implementation of a program. A test requirement is *covered* when there is at least one test case that exercises it. The testing techniques can be classified into three main types: *mutation*, *functional* and *structural* techniques [15]. Figure 1 shows this classification.

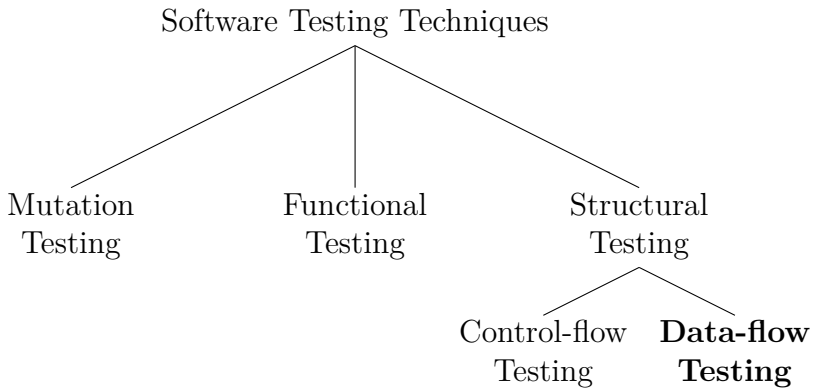


Figure 1 – Software testing techniques classification

A *testing criterion* comprises a set of requirements that should be verified to achieve a certain level of testing. A test set is considered *adequate* to a given criterion when all of the criterion requirements are covered. The usefulness of a testing criterion is two-fold. It supports the development of test cases by indicating what should be exercised by them. It also provides a quality assessment of the test cases already developed.

The mutation technique is based on the competent programmer hypothesis. That is, programmers write programs that are very close to the correct version, with the possible exception of few defects. The mutation test consists of inserting known flaws in the program source code (or object code) to generate faulty versions; each faulty version is called a

mutant program [15]. A test set that contains test cases that produce different outputs in the original and mutant programs is likely to reveal real faults present in the code.

The idea underlying functional testing is to identify features that a program should carry out and create a test to check whether these features are correct. Examples of functional testing criteria are *Equivalence Class Partitioning* and *Boundary-value Analysis* [15].

Structural testing analyzes the source or object code of a program to determine the requirements that should be covered by a test set. This kind of testing has requirements associated with the implementation of the program. For instance, a criterion can require that all the lines of a program should be executed. In this case, the lines are the requirements, and a test set T is considered adequate to all-lines criterion if every line of the program is executed at least once by test cases belonging to T.

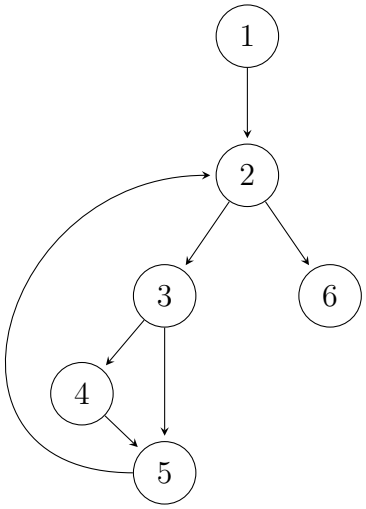
As the goal of this research is related to structural testing, in the following sections we describe it in detail. We start off by describing structural testing in general terms in Section 2.2. The concepts needed and the definition of structural testing criteria are formally presented in Section 2.3.

2.2 Structural testing

Structural testing is a technique that utilizes the software implementation to derive the testing requirements that should be exercised by a test set [15].

```
int max(int array [], int length)
/* 1 */ {
/* 1 */ int i = 0;
/* 1 */ int max = array[++i];
/* 2 */  while (i < length)
/* 3 */   {
/* 3 */    if (array[i] > max)
/* 4 */       max = array[i];
/* 5 */    i = i + 1;
/* 5 */   }
/* 6 */   return max;
/* 6 */ }
```

(a) Example program



(b) CFG of the example program

Figure 2 – Example program and its control-flow graph (CFG)

Structural testing can be divided into two groups — *control-* and *data-flow* testing. The CF testing requirements are derived from a flow graph obtained from a program — the *control-flow graph* (CFG). *Nodes* in a CFG are associated with a sequence of program

statements, and *edges* are associated with the possible transfer of control between nodes. All the statements associated with a node are executed sequentially so that the execution of the first statement implies the execution of all statements belonging to the node.

Figure 2a presents an example program which determines the maximum element in an array of integers. The number inserted inside the comments refers to the node each statement is associated with. Figure 2b presents the CFG obtained from the example program of Figure 2a.

The CFG can be extended to include information regarding the DF of the program. When a variable receives a new value, it is said that a *definition* has occurred. For instance, there is a definition of variable i at node 5 (regarding statement $i = i + 1;$) of the example program. A *use* of a variable happens when its value is referred to. A distinction is made between a variable referred to compute a value and to compute a predicate. When referred to in a predicate computation, it is called a *p-use* and is associated with edges; otherwise it is called a *c-use* and is associated with nodes.

Below we describe in general terms some control- and data-flow testing criteria:

Table 1 – Entities required by the all-nodes and all-edges criteria

All nodes	All edges
1	(1,2)
2	(2,3)
3	(2,6)
4	(3,4)
5	(3,5)
6	(4,5)
	(5,2)

All-nodes: a test set should cover all nodes of the CFG.

All-edges: a test set should cover all edges of the CFG. A test set that is adequate to this criterion is adequate to the all-nodes criterion as well.

All-uses: this criterion involves the development of tests which exercise every value assigned to a variable and its subsequent references (uses) occurring either in a computation or in a predicate. These requirements are called *definition-use associations* (dua).

Table 1 shows requirements for the all-nodes and all-edges criteria and Table 2 for the all-uses criterion of the example program. Nodes are identified by its label in the CFG; edges by a tuple (n', n), which represents an edge between node n' and node n; Duas are identified by a triple (d, u, X) where X is a variable, d is a node containing a definition of

Table 2 – Entities required by the all-uses criterion

All uses		
(1, 6, max)	(4, 6, max)	(1, (3,4), max)
(1, (3,5), max)	(4, (3,4), max)	(4, (3,5), max)
(1, 4, i)	(1, 5, i)	(1, (2,3), i)
(1, (2,6), i)	(1, (3,4), i)	(1, (3,5), i)
(5, 4, i)	(5, 5, i)	(5, (2,3), i)
(5, (2,6), i)	(5, (3,4), i)	(5, (3,5), i)
(1, 4, array)	(1, (3,4), array)	(1, (3,5), array)
(1, (2,3), length)	(1, (2,6), length)	

X and u is a node (edge) containing a c-use (p-use) of X. In the next section, we formalize the concepts presented in this section.

2.3 Formal definitions

Now we introduce the formal definitions and the flow graph concepts based on a simple formal programming language. These definitions are an extension of the definitions presented in Rapps and Weyuker [3, 4]. The language allows only simple variables, which comprises scalar variables but not aggregate variables. The following are *valid statement* types:

Input statement: read $x_1, \ldots, x_n$
where $x_1, \ldots, x_n$ are variables.

Assignment statement: $y := f(x_1, \ldots, x_n)$
where f is an n-ary function ($n \geq 0$) and $y, x_1, \ldots, x_n$ are variables.

Output statement: print $e_1, \ldots, e_n$
where for each $i = 1, \ldots, n$, e_i is either a literal or a variable.

Unconditional transfer statement: goto i
where i is a unsigned integer.

Conditional transfer statement: if $p(x_1, \ldots, x_n)$ then goto i
where p in an n-ary predicate ($n > 0$), $x_1, \ldots, x_n$ are variables and i is an unsigned integer. 0-ary predicates are prohibited.

Halt statement: stop.

A *program* is a finite sequence of valid statements $(s_1, \ldots, s_i, s_{i+1}, \ldots, s_n)$ of length n, each statement s_i is identified by its index i in the sequence. A program contains exactly one start statement $s = s_1$ (the first statement from the sequence) and at least one halt

statement. The last statement (s_n) from the sequence must be either a halt statement or an unconditional transfer. For every transfer statement *goto i* or *if p then goto i*, i must be the index t, $1 \leq t \leq n$, of some statement in the sequence. This statement is the *target* of the transfer statement. Note that we use the term *transfer statement* when we wish to include both conditional and unconditional transfer statements.

We say that s_i *physically precedes* s_j (s_j *physically succeeds* s_i) if and only if $s_j = s_{i+1}$. That is, s_j is the exactly next element after s_i in the sequence. The statement s_i *executionally precedes* statement s_j (s_j *executionally succeeds* s_i) if and only if:

1. s_i is a transfer statement and s_j is its target; or

2. s_i is not a halt statement or a unconditional transfer statement and s_i *physically precedes* s_j.

A statement s is *syntactically reachable* if and only if there is a sequence of statements $(s_1, \ldots, s_j)$ such that $s = s_j$ and for each $i = 1, \ldots, j-1$, s_i executionally precedes s_{i+1}. Statements that are not syntactically reachable are known as *dead-code*. A transfer statement is *ineffective* if it physically precedes its target. All other transfer statements are *effective*. Dead-code and ineffective transfer statements are not allowed. This way, every statement in the program is syntactically reachable and all conditional transfer statements have two different statements that executionally succeeds it.

A program can be decomposed into a set of disjoint blocks of statements such that once the first statement in the block is executed all statements are executed in sequence. A *block* is the maximal set of ordered statements $b = (s_1^b, \ldots, s_n^b)$ such that, if $n > 1$ then, for each $i = 1, \ldots, n-1$, s_i^b is the unique executional predecessor of s_{i+1}^b and s_{i+1}^b is the unique executional successor of s_i^b. The first statement of a block is the unique that might have an executional predecessor outside the block. Similarly, the last statement in a block is the unique statement that might have an executional successor outside the block. Note that the first statement of the block is the only one that may be executed after the execution of a statement in another block. Furthermore, since effective conditional transfers always have two different executional successors, every conditional transfer must be the last statement of a block.

Let P be a program mapped into a flow graph $G(N, E, s, e)$ where N is the set of nodes (a node corresponding to each block), s is the start node (a node whose the corresponding block includes the program's start statement), e is the set of exit nodes (nodes which the corresponding blocks includes a halt statement), and E is the set of edges where each edge (n', n) represents a possible transfer of control between node n' and node n, i.e., the last statement of n' is not an unconditional transfer and it physically

precedes the first statement of n, or the last statement of n' is a transfer whose target is the first statement of n.

A *path* is a finite sequence of nodes (n_i, ..., n_k, n_{k+1}, ..., n_j), where $i \leq k < j$, such that $(n_k, n_{k+1}) \in E$ for each $k = i, \ldots, j-1$. A node n_k is said to be a *predecessor* of a node n_{k+1}, and n_{k+1} is a *successor* of n_k, if there exists an edge (n_k, n_{k+1}) in E. A *complete path* in $G(N, E, s, e)$ is a path (n_i, ..., n_j) whose $n_i = s$ and $n_j \in e$. A *simple path* is a path in which no node occurs twice in the path except possibly the first and the last nodes. A *loop-free path* is a path (n_i, ..., n_j) such that $n_k \neq n_l$ for $k \neq l$.

DF testing requires that selected test cases exercise paths in a program between every point a value is assigned to a variable and its subsequent references. A *definition* of a variable X occurs when X is in the left-hand side of an assignment statement or is a parameter of an input statement. A *c-use* of a variable X happens when X occurs in the right-hand side of an assignment statement or is a parameter of an output statement. A *p-use* of X takes place when X is part of a predicate. A *definition-clear* path with respect to (wrt) a variable X is a path where X is not redefined in any node in the path, except possibly in the first and last ones. Figure 3 describes the CFG of the example program annotated with the set of definitions (def), c-uses, and p-uses associated with nodes and edges.

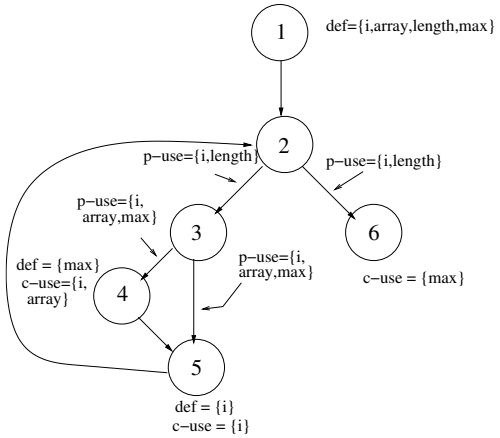


Figure 3 – Annotated flow graph of the example program

DF testing criteria in general requires that *definition-use associations* (duas) be covered. The triple $D = (d, u, X)$, called c-use dua, represents a DF testing requirement involving a definition at node d and a c-use at node u of variable X such that there is a definition-clear simple path wrt X from d to u. Likewise, the triple $D = (d, (u', u), X)$, called p-use dua, represents the association between a definition and a p-use of a variable X. In this case, a definition-clear path (d, ..., u', u) wrt X should exist.

A test set T is a set of test cases $(t_1, t_2, \ldots, t_n)$ of a program P. Every test case t_i has an associated complete path p_i in the CFG of the program P. So, a test set T can be represented by a set of complete paths $(p_1, p_2, \ldots, p_n)$.

2.3.1 Control-flow criteria

We now introduce the formal definition of CF criteria. Let P be a program mapped into a flow graph $G(N, E, s, e)$ and T a test set of program P.

All-nodes: T satisfies the all-nodes criterion if and only if $\forall n \in N, \exists p \in T$ such that p includes n. That is, T includes every node from the CFG.

All-edges: T satisfies the all-edges criterion if and only if $\forall (u', u) \in E, \exists p \in T$ such that p includes a node $n_i = u'$ and $n_{i+1} = u$. That is, every $(u', u) \in E$ is included in T. An edge (u',u) is covered if for some test case, u is executed immediately after u'.

All-paths: T satisfies the all-paths criterion if and only if T includes every complete path of G. Due to loops, many graphs have an infinite number of complete paths.

2.3.2 Data-flow criteria

We present below the formal definition of all-defs, all-uses, and all-du-paths DF criteria. Let P be a program mapped into a flow graph $G(N, E, s, e)$, $Def(n)$ the set of variables for which node n includes a definition. T is the test set of program P.

All-defs: Let $Dua(d, X)$ be the set of duas such that involves a definition of variable X at node d. The all-defs DF testing criterion requires that for every node $n \in N$ and every variable $v \in Def(n)$ the set of paths executed by a test set T to include a definition-clear path for some dua in $Dua(n, v)$. A test set with such a property is said to be *adequate* to the all-defs criterion for program P since all required duas were *covered*.

All-uses: The all-uses DF testing criterion requires the set of paths executed by the test cases of a test set T to include a definition-clear path for each dua (d, u, X) or (d, (u', u), X) of a program P. A test set with such a property is said to be *adequate* to the all-uses criterion for program P since all required duas were *covered*.

All-du-paths: The all-du-paths DF testing criterion requires the set of paths executed by the test cases of a test set T to include every definition-clear simple path for each dua (d, u, X) or (d, (u', u), X) of a program P. Note that there might exist more than one path that should be exercised for each dua. Since the path is simple, no node occurs twice on the path except possibly the first and the last. A test set with such a property is said to be *adequate* to the all-du-paths criterion for program P since all required duas were *covered*.

2.4 Final remarks

In this chapter, we presented the software testing concepts related to this research. Structural testing — especially control- and data-flow testing criteria — concepts were presented in detail. In the next chapter, we examine the related work.

3 Related work

In this chapter, we discuss the different strategies and tools to track structural testing requirement at run-time.

3.1 Instrumentation strategies for control-flow

The need for efficient program instrumentation algorithms is not new. The search for efficient instrumentation is old as the first profilers. Profiling tools such as PurifyPlus[1] instrument the object code to collect information like the number of times a statement is executed. This information allows one to obtain the coverage regarding the statements executed in a test; that is, the all-nodes criterion coverage [3, 4]. However, profiling tools impose program slowdown and memory overhead.

Different strategies have been proposed to tackle these issues. Ball and Larus [16] and Agrawal [17] have defined algorithms that analyze the code to better locate the instrumentation code. The information collected at these locations allows the inference of the total coverage. As a result, less instrumentation code is required and the slowdown and memory overhead are reduced.

Tikir and Hollingsworth [18] have introduced two mechanisms to improve program instrumentation to monitor program blocks (nodes). The first one works by instrumenting the program as it runs. When a method[2] is first visited, it is instrumented. The second mechanism involves removing the instrumentation code of blocks already covered. At determined time intervals, the covered blocks are verified and the additional inserted code is then removed. These mechanisms implement a *demand-driven* approach.

Instrumentation issues have also been addressed by exploiting hardware features and sampling. Soffa et al. [19] examine the use of the *Last Branch Record* (LBR) — a hardware mechanism that reports the last executed branches — to capture the edge coverage at hardware level. Initial investigation suggests that the memory increment is reduced and the slowdown imposed is dependent on the sampling rate of the LBR. Fischmeister and Ba [20] developed algorithms to determine the optimal sampling and instrumentation so that the accuracy/overhead relation is maximized.

The works aforementioned were targeted to track CF requirements at run-time. Comparatively, few works address efficient instrumentation for dua coverage. In the next section we present the main strategies to track DF requirements.

---

1 <http://www.ibm.com/software/awdtools/purifyplus/>

2 We utilize *method* as any unit of the program (e.g., procedure, function, method).

3.2 Instrumentation strategies for data-flow

According to Santelices and Harrold [13] the most common approach to track DF testing requirements is a strategy known as *last definition*. Misurda et al. [12] describes a demand-driven last definition approach. We present this demand-driven strategy in Section 3.2.2. To tackle the expensive cost of dua monitoring Santelices and Harrold [13] designed an efficient strategy for the last definition method. This approach is described in Section 3.2.3. The first attempt to track duas was introduced in the ASSET tool [21] — its strategy is also presented in the next section.

3.2.1 ASSET

Frankl and Weyuker [21] developed a tool, called ASSET (*A System to Select and Evaluate Tests*), to support the application of their family of DF testing criteria. The ASSET strategy utilizes finite state automata to track duas. Each dua $D = (d, u, X)$ or $D = (d, (u',u), X)$ is mapped into an automaton. When a node is visited, every automaton associated with a dua D is checked whether it has reached the final state. In this case, the dua is set as covered and the automaton is not walked through anymore. Basically, for a c-use dua, the automaton has three states. Figure 4 describes this automaton.

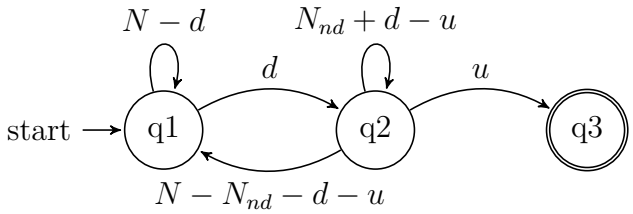


Figure 4 – Finite automaton associated with a c-use dua

The c-use automaton remains in the first state until the program execution reaches the node d. As soon as the node d is reached the automaton goes to the second state. In the second state the automaton might go to the final state, when the program execution reaches node u, or might return to first state when the program execution reaches a node n, such that $n \neq d$ and n has a definition of X. The automaton remains in the second state otherwise.

In Figure 4 and 5, N is the set of all nodes from the CFG and N_{nd} is the set of all nodes without a definition of X. A path is traversed and the automaton receives every node visited. If the automaton reaches the final state then program execution visited node u, after visiting node d, without passing through a node that redefines X.

For p-use duas $D = (d, (u',u), X)$ the automaton is slightly different, the automaton has four states. Figure 5 describes this automaton.

The automaton of Figure 5 remains in the first state until the program execution reaches node d. As soon as node d is reached the automaton goes to the second state. In the second state the automaton might go to the state $q3$ when the program execution reaches node u', or might return to first state when the program execution reaches a node n, such that $n \neq d$ and n has a definition of X. The automaton remains in the second state otherwise. In the third state, the automaton might go to the final state when the program execution reaches node u, or might return to first state, when the program execution reaches a node n that redefines X and $n \neq d$, $n \neq u$. The automaton returns to the state $q2$ otherwise.

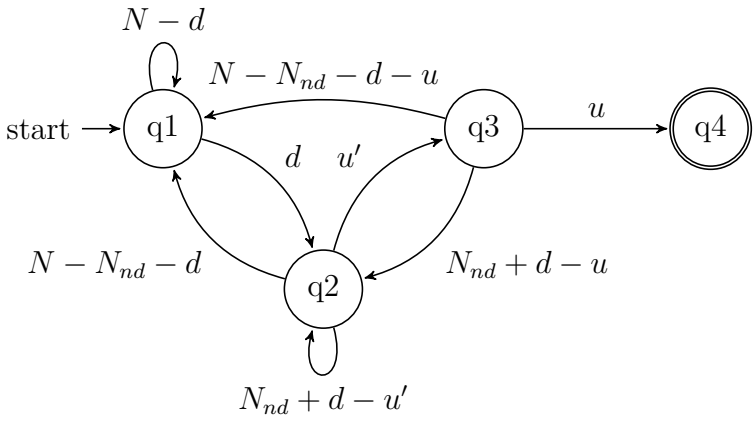


Figure 5 – Finite automaton associated with a p-use dua

The automaton described in Figure 5 guarantees that final state is reached if and only if a definition-clear path (d, ..., u', u) wrt X was traversed. Note that from the definition of a dua in Section 2.3 we know that $u' \in N_{nd}$.

3.2.2 Demand-driven

Misurda et al. [12] utilizes a *demand-driven* strategy to instrument programs to track down duas. The idea is to instrument the object code at initial points (called seeds). The seeds are probes inserted at nodes d in which there are definitions of variable; they are called *def probes* and one def probe is inserted at d if there exist at least one dua D = (d, u, X) (or D = (d, (u',u), X)). When a def probe is reached, d is recorded as the *last definition* of X and a respective *use probe* for every D is inserted at u. A use probe, when reached, checks whether the last definition is d; if true, it records the coverage of the respective dua and is removed. To track p-use duas (e.g., D = (d, (u',u), X)), probes are inserted to record the last node visited; they are only covered if d is the last definition of X and u' is the previous node visited.

For example, consider the example program from Figure 2a (we simplified the problem to monitor c-use dua only). The algorithm proposed by Misurda et al. firstly

builds the flow graph (Figure 3) and determines the duas (Table 2). Then, the algorithm proceeds in two phases in which it:

1. Iterates over all duas to group definitions and uses; and
2. Constructs a data-structure, called *Probe Location Table* (PLT), to conduct the demand-driven instrumentation.

Table 3 describes the group of definitions and uses determined by the first phase of the algorithm. Each row corresponds to a node n, the second column identifies whether the node is seed or not, the third and the fourth columns describe which variables are defined and used, respectively. The last column describes pairs (u, X) such that u is a node in which a probe will be placed to track a use of variable X (use probe).

Table 3 – Group of definitions and uses for each node

Node	Seed	Definitions	Uses	Place Uses
1	Yes	{i, array, length, max}	-	(6,max), (4,i), (5,i), (4,array)
2	No	-	-	-
3	No	-	-	-
4	Yes	{max}	{i, array}	(6, max)
5	Yes	{i}	{i}	(4, i), (5,i)
6	No	-	{max}	-

To exemplify how the demand-driven algorithm works, consider that path (1, 2, 3, 4, 5, 2, 6) has been traversed in the flow graph from Figure 3. Before the execution starts, probes at nodes 1, 4 and 5 should be added statically (these are seed probes). When node 1 is visited its probe records variables {i, array, length, max} as last defined at node 1. The probe also insert use probes at nodes 4, 5 and 6 (nodes that contain a use for variables defined at node 1). Note that nodes 4 and 5 have already a probe (seed probes), then def probe at node 1 skips inserting use probes for this nodes.

When node 2 and 3 are visited, there is nothing to do (these nodes do not have probes). At node 4, duas (1, 4, i) and (1, 4, array) are recorded as covered, variable *max* is recorded as last defined at node 4 and a use probe should be inserted at node 6 (this last step is also skipped, since node 6 has already a probe inserted by node 1). At node 5, dua (1, 5, i) is recorded as covered. It is not needed to insert probes at nodes 4 and 5 (these nodes already have probes). This probe also records variable i as last defined at node 5.

Finally, when node 6 is traversed, dua (4, 6, max) is recorded as covered. Since this dua was covered, the use probe inserted to track it should be removed. Note that a def probe must remain until all reachable uses are covered. The tool Jazz [22] utilizes this approach.

3.2.3 Matrix-based

Santelices and Harrold [13] proposed a *matrix-based* strategy. This mechanism is able to precisely track duas at run-time. They create a coverage matrix m initialized with zeroes where each column corresponds to a use and each cell in a column represents a definition for that use. Although not all cells correspond to a dua, this approach allows quick access of the matrix by using two indexes — a use ID and a definition ID. At run-time, probes track the ID of the last executed definition for that variable. At each use, the instrumenting inserts code that reads the last definition ID and uses it as the row index when accessing the corresponding cell in the matrix. The tool DUA-FORENSICS [13] uses this approach.

3.3 Structural testing tools

There are a number of tools that perform structural testing of programs. Since we are interested in developing a tool to support DF testing coverage of programs compiled into Java bytecodes, we revised a wide variety of well-known Java testing coverage tools.

In the next sections, we present the following tools: JaCoCo [23], Clover [24], Cobertura [25], EMMA [26], CodeCover [27], Quilt [28], JVMDI [29] and GroboCoverage [30]. However, none of these tools supports DF testing; that is, they are unable to track duas and determine the dua coverage. These tools were targeted to track CF requirements at run-time.

A limited number of DF testing tools are available publicly and hardly used in real settings. In other words, the tools available to carry out DF coverage are not scalable. Since fewer tools tracks DF testing requirements, our revision of DF testing tools comprises tools targeted to Java and other languages as well.

We reviewed the following DF tools: ASSET [21], TACTIC [31], ATAC [32], POKE-TOOL [33], JaBUTi [34], DUA-FORENSICS [13] and DFC [35]. The following Eclipse[3] plug-ins that support DF testing was reviewed: Jazz [22], InsECTJ [36] and Coverlipse [37]. The mocking tool JMockit [38] that supports CF testing and also a kind of DF testing coverage is also revised. Table 4 lists the features of the tools analyzed in the next sections.

3.3.1 JaCoCo

JaCoCo (*Java Code Coverage*) [23] is an active open-source project that performs Java program code coverage. JaCoCo uses ASM[4] to instrument object code and classes on-the-fly. The ASM library dependency is not a problem since the library is included

[3] <http://www.eclipse.org/>
[4] <http://asm.ow2.org/>

with the tool and an isolation of the classes avoids conflicts. This trick also allows JaCoCo to be verified by itself.

Classes instrumented by JaCoCo increase their size in total about 30% and since probe execution does not require any method calls, only local instructions, the estimated overhead is less than 10%. JaCoCo uses boolean arrays to monitoring CF requirements, similarly to EMMA, but the instrumentation is quite different. JaCoCo inserts probes to track edges, instead of nodes. These probes infer the total coverage of instructions and edges. As consequence, less probes are needed and the slowdown and memory overhead are reduced.

3.3.2 Clover

Clover [24] is a well-known comercial coverage tool that carries out statement and edge coverage of Java programs. Clover instruments the source code to collect code coverage. This tool can do per-test coverage; that is, it is possible to know which tests cover a particular class, method or line of code. Clover has plug-in version to Eclipse and IntelliJ[5]. As a closed-source tool, we could not obtain details about the internal structure used to collect coverage information.

3.3.3 Cobertura

Cobertura [25] uses the ASM library to instrument Java object code. It alters Java classes by adding calls to a run-time routine in every node of a method's flow graph. Thus, Cobertura monitors either nodes or edges. However, it has run-time library dependencies such as Log4j[6]. These run-time dependencies may muddle some programs to be analyzed by the tool; for example, if the program under test has the same library dependency.

3.3.4 EMMA

EMMA [26] is an efficient open-source toolkit for measuring Java code coverage. The run-time overhead imposed by classes instrumentation is small (5-20%). Besides, it is written completely in Java without external library requirements and works with legacy Java Virtual Machines. However, EMMA supports only class, method, line and block (node) coverage.

EMMA instrumentation modifies Java classes at run-time (an on-the-fly approach) when the classes are loaded by the class loader or statically, as a second phase compilation. EMMA instrumentation inserts a boolean matrix as a field for each Java class where each row in the matrix corresponds to a method and each cell in a row corresponds to a node in

[5] <http://www.jetbrains.com/idea/>
[6] <http://logging.apache.org/log4j/>

the flow graph. Every time a node in a method's flow graph is reached the corresponding cell in the matrix is set to true.

3.3.5 CodeCover

CodeCover [27] instruments Java source code to monitor statments and edges. The edge coverage does not take into loop statements since it implements a loop criterion. Loop coverage has at most three coverable items representing the number of iterations: zero times, one time and more than one time. Moreover, edges created by possible exceptions are ignored.

3.3.6 Quilt

Quilt [28] is a Java code coverage tool that intercepts code as it is loaded by the class loader (on-the-fly) and alters it (instrument). Quilt manipulates object code and supports block (node) and edge coverage. Quilt inserts probes that increments counters whenever a block of code or edge is executed. A coverage report is generated at the end of the test suite execution.

3.3.7 JVMDI

JVMDI *Code Coverage Analyser* performs code coverage using JVMDI (*Java Virtual Machine Debug Interface*)[7]. This small utility works as follows: when the shared library is loaded into a Java Virtual Machine (1.4 or newer) all lines of code executed will be recorded. This is a relatively weak coverage method, but this method enables to identify regions (lines of code) which have not been adequately tested.

3.3.8 GroboCoverage

GroboCoverage [30] instruments the object code using BCEL[8]. The instrumented object code contains calls to GroboCoverage run-time routine (called logger). The run-time routine notifies which class, method and probe indexes are covered. During the object code instrumentation phase, a set of data files are generated informing how method and probe indexes are translated into the source code file. The reporting phase combines the data collected at run-time with the data generated statically (in the instrumentation phase) to generate XML reports, and from that a HTML report. GroboCoverage supports coverage of: lines, instructions, edges and methods.

[7] <http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jvmdi-spec.html>
[8] <http://commons.apache.org/proper/commons-bcel/>

3.3.9 TACTIC

Ostrand and Weyuker [31] describe TACTIC (*Test Analysis and Coverage Tool, Intended for C*), a prototype tool for DF testing coverage of C programs with single-level pointer references. TACTIC is divided in three main components: *pre-execution analyzer*, *execution monitor* and *user interface*. Pre-execution analysis is responsible for performing analysis in the source code, computing DF testing requirements and instrumenting the program source code.

This component proceeds as follows. An abstract syntax tree is built from source code. Transformations in the syntax tree are performed to make DF analysis easier. The syntax tree is then de-parsed back into source code and duas are computed from the modified source code. The algorithms for computing duas carry out program-point-specific alias computations, for one level of pointer indirection. Finally, instrumentation is done by compiling the transformed source code with calls to the *execution monitor*. TACTIC tracks memory positions to determine precisely which duas were covered.

3.3.10 ATAC

Horgan and London [32] describe ATAC (*Automatic Test Analysis for C*), a tool for DF coverage testing of C programs. The ATAC preprocessor parses C source code and creates an abstract sytax tree in-memory for each C function. An annotated flow graph with variable definitions and uses is created from this syntax tree. To create an instrumented version of the source code a mark is inserted in the syntax tree for each node in the flow graph.

By de-parsing the marked syntax tree, it is possible to create an instrumented version of the source code with calls to the ATAC run-time routine. Each call to it receives as argument a node identification and a context pointer. The context contains def/uses tables (generated from the flow graph), information about testing requirements already covered and a function nesting level (to handle recursive calls).

ATAC creates during the instrumentation phase a file with static information about testing requirements that might be covered during test cases execution. To create this file the annotated flow graph is analyzed. The ATAC run-time routine monitors DF testing requirements as follow. The def/uses tables contains information about which variables are defined and used in a given block. ATAC keeps track, for each variable, the block at which it was defined. When a block that uses a defined variable is reached the run-time routine verifies the last definition of that variable and the corresponding dua is recorded in the trace file.

3.3.11 POKE-TOOL

POKE-TOOL (*Potential Uses Criteria TOOL for program testing*) [33] was developed to support DF testing using the *Potential Uses* criteria family [39]. This tool also supports other testing criteria such as all uses, all nodes and all edges.

From the analysis of program source code, POKE-TOOL generates an annotated flow-graph with variable definitions and uses for each method. After analysis, it instruments the program source code. POKE-TOOL is a multilanguage tool supporting programs written in several languages (currently POKE-TOOL accepts programs written in C, COBOL and FORTRAN). POKE-TOOL strategy to track duas is similar to that of ASSET.

3.3.12 JaBUTi

JaBUTi (*Java Bytecode Understanding and Testing*) [34] is a tool for structural testing of Java programs that supports control- and data-flow criteria. Furthermore, JaBUTi analyzes object code (bytecode) to determine the testing requirements; thus, source code analysis is not needed. JaBUTi instruments object code using the BCEL library. The instrumented code records in the RAM memory the path (trace) traversed during a test suite execution. The trace is then post-processed to determine the covered requirements. The recorded trace should not be too long; otherwise, the instrumented program may run out of memory.

3.3.13 DFC

Bluemke and Rembiszewski [35] describe the DFC (*Data Flow Coverage*), A testing tool for Java programs implemented as an Eclipse plug-in. DFC finds all duas in a method and provides a visual flow graph annotated with variable definitions and uses. Thus, DFC duas are intra-procedural. Since Java is object-oriented, a method might contain references to class attributes and fields. So, definitions of fields and class attributes are located in the first node of the flow graph.

DFC is divided in five modules. *Knowledge base* module analyzes the source code to generate a list of classes and methods. *Instrumentation* module inserts additional instructions in the source code to send DF information to DFC. This module also builds an annotated flow graph with variable definitions and uses for each method. *Visualization* module uses the flow graph built by the *Instrumentation* module to display the graph on the screen. *Requirements* module is responsible for extracting the duas from the flow graph. *Analyzing* module determines the duas that are covered.

3.3.14 InsECTJ

InsECTJ [36] is another Eclipse plug-in that uses the BCEL library. It allows users to define instrumentation tasks. The key ideia is to implement *monitors* at particular events. Duas can thus be monitored by associating monitors at definition and uses of variables.

3.3.15 Coverlipse

Coverlipse [37] is an Eclipse plug-in that uses the BCEL library to instrument the object code and supports all nodes and all uses criterion. Unfortunately, DF testing is not precise (p-uses duas are not monitored and c-uses are not accurate). Futhermore, this tool is not in active development.

3.3.16 JMockit

JMockit [38] is a testing tool to create mock objects. Besides, JMockit also provides three different code coverage metrics: line coverage, path coverage, and data coverage. Line coverage metric takes into account both statements and edges.

JMockit path coverage determines possible paths through the implementation body and monitors these paths from method entry to normal method exit. On the other hand, data coverage checks if instance and static non-final fields were fully exercised. A field is fully exercised when its last assigned value is referenced. So the data coverage in JMockit is the all-defs criterion for fields only. It is a inter-procedural criterion since a field might be referenced in any method.

3.4 Discussion

Table 4 lists the features of the tools analyzed in the previous sections. Each row corresponds to a tool; the first column identifies the tools, columns 2 to 7 contain the features found in the testing tools; the eighth and ninth columns indicate the type of instrumentation and the date of last update, respectively. The check mark indicates that the tool supports the feature.

The main features investigated are related to the type of coverage provided by the tools. Thus, columns 2 to 5 indicate whether the tools provide line, node, edge or dua coverage, respectively. Column 6 informs if a tool is open- or closed-source; and column 7 indicates if the tools are integrated to an Interactive Development Environment (IDE) such as the Eclipse environment.

The type of instrumentation is divided into five groups: source, object, debug, Soot [40] and JVM, being of type *source* when the source code is instrumented, *object*

Table 4 – Structural testing tools

Tool name	Line coverage	Node coverage	Edge coverage	Dua coverage	Closed-source	IDE integration	Instrumentation type	Last update
JaCoCo	✓	✓	✓			✓	Object	May-2014
Clover	✓	✓	✓		✓	✓	Source	Jul-2014
Cobertura	✓	✓	✓			✓	Object	Aug-2013
EMMA	✓	✓				✓	Object	Jun-2005
CodeCover	✓	✓	✓			✓	Source	Mar-2011
Quilt	✓	✓	✓				Object	Oct-2003
JVMDI	✓						Debug	Apr-2002
GroboCoverage	✓	✓	✓				Object	Nov-2004
ASSET				✓	✓		Source	-
TACTIC				✓	✓		Source	-
ATAC	✓	✓	✓	✓			Source	-
POKE-TOOL	✓	✓	✓	✓			Source	-
JaBUTi	✓	✓	✓	✓			Object	-
DUA-FORENSICS			✓	✓			Soot	May-2013
DFC				✓	✓	✓	Source	-
Jazz	✓	✓	✓	✓	✓	✓	JVM	-
InsECTJ		✓	✓	✓		✓	Object	Nov-2011
Coverlipse	✓	✓		✓		✓	Object	Fev-2009
JMockit	✓	✓	✓	✓			Object	Jul-2014

when the object code is modified, *debug* if a monitor interface (as in JVMDI) is used, *Soot* when an intermediate language representation is necessary, and, finally, *JVM* if the instrumentation depends on the execution environment.

From Table 4, we note that 16 out of 19 tools support some CF criteria (all-lines, all-nodes or all-edges) and 11 out of 19 support some kind of DF testing. Thus, the difference is not big because structural testing can only be carried out with tool support. As a result, ASSET, TACTIC, ATAC, POKE-TOOL, JaBUTi, DUA-FORENSICS, DFC, Jazz and InsECTJ were developed at academic and research settings to support DF testing. To the best of our knowledge, the only two tools independently developed to support DF testing are Coverlipse and JMockit. Table 5 summarizes our comparison of the DF testing tools. It lists whether a tool tracks inter- or intra-procedural duas or both, the technique it relies on to monitor duas, the language it supports, and its availability.

Four tools supports inter-procedural duas coverage: JaBUTi, DUA-FORENSICS, InsECTJ and JMockit. DUA-FORENSICS requires a single entry point while JaBUTi

Table 5 – DF testing tools

Tool name	Coverage	Technique	Language	Availability
ASSET	Intra	Automata	Pascal	
TACTIC	Intra	Memory tracking	C	
ATAC	Intra	Last definition	C/C++	✓
POKE-TOOL	Intra	Automata	C	✓
JaBUTi	Inter/Intra	Last definition	Bytecode	✓
DUA-FORENSICS	Inter/Intra	Matrix-based	Java	✓
DFC	Intra	—	Java	
Jazz	Intra	Demand-driven	Java	
InsECTJ	Inter/Intra	User-defined	Java	✓
Coverlipse	Intra	Path recording	Java	✓
JMockit	Inter	—	Java	✓

restricts the level of methods invocations to determine inter-procedural requirements. JMockit implements a simplified version of the all-defs criterion [3, 4] for instance and static non-final fields. In general, the tools implements the techniques devised by its authors to monitor duas. The exceptions are DFC, Coverlipse and JMockit for which we could not determine the dua tracking technique utilized; though, Coverlipse records the traversed path which is then processed to determine the covered duas. The more recent tools support Java while the older ones focus on procedural languages, especially C. Finally, seven tools are available for download.

As already noticed by Yang et al. [41], and corroborated by our own survey, there is not a commercial tool supporting DF testing. Commercial (e.g., Clover) and open-source (e.g., Cobertura, JaCoCo) CF testing tools are available and used to assess control coverage of large systems [7]. Hassan and Andrews [42] report that they were able to find only two working DF testing tools: ATAC and DUA-FORENSICS. Even these two tools had restrictions in analyzing particular programs. These authors suggest that the high cost associated with tracking duas has discouraged tool vendors from implementing it. These are evidences that the current strategies to track duas are not able to tackle large and/or long running programs.

3.5 Final remarks

In this chapter, we discussed the different strategies and tools to track at run-time structural testing requirements. Since our goal is to provide efficient DF coverage, we introduced in detail techniques to determine the coverage regarding DF criteria.

From our investigation of the available testing tools, there are not DF testing tools available to the practitioner. This situation is explained in part due to the lack of strategies to track DF testing requirements that scale for large and/or long running programs. In

the next chapter, we present a novel technique — the Bitwise Algorithm (BA) — to track duas targeted to this class of programs.

4 Bitwise dua coverage algorithm

In this chapter, we present a new algorithm to track intra-procedural definition-use associations (duas) at run-time. Its main characteristic is to utilize efficient bitwise operations and inexpensive data structures to track duas. We also present cost analyses regarding RAM memory requirements and run-time costs.

4.1 Algorithm description

The Bitwise Algorithm (BA) for dua coverage was recently proposed [14] to reduce run-time costs of monitoring duas. BA is based on sets of duas associated with each node of the flow graph. We start off BA presentation by describing these sets.

Let us say node 5 of the flow graph of Figure 3 has just been visited. This implies that duas (5, 4, i), (5, 5, i), (5, (2,3), i), (5, (2,6), i), (5, (3,4), i) and (5, (3,5), i) may be covered in the future, provided the respective use is reached without a redefinition of i. We say that these duas are *born* at 5 due to the definition of variable i. On the other hand, duas (1, 4, i), (1, 5, i), (1, (2,3), i), (1, (2,6), i), (1, (3,4), i) and (1, (3,5), i) are *disabled* at node 5 since variable i was redefined.

A set of p-use duas, called *sleepy duas*, is also associated with each node. Consider that node 3 has just been visited. The next node to be visited may be node 4 or node 5. Thus, the p-use duas enabled to be covered at the next visited node are (1, (3,4), max), (4, (3,4), max), (1, (3,4), i), (5, (3,4), i), (1, (3,4), array), (1, (3,5), max), (4, (3,5), max), (1, (3,5), i), (5, (3,5), i) and (1, (3,5), array). These duas have in common edges in which node 3 is the origin node. All the other p-use duas are *temporarily disabled* until node 4 or 5 is subsequently traversed.

The set of *sleepy* duas associated with node 3 is comprised of these temporarily disabled duas. Its usefulness is to filter out those p-uses that cannot be covered at the immediate next nodes. Thus, the sleepy duas set of node 3 can be used to rule out p-use duas that cannot be covered at 4 or at 5, the immediate next nodes.

For some nodes, the sleepy duas set comprises all p-use duas. The sleepy duas associated with node 4 encompass all p-use duas. As there is no p-use dua with an edge (4, u), no p-use duas can be covered when node 5 is visited after the traversal of node 4.

The last set of duas associated with the nodes of a flow graph is the set of *potentially* covered duas. Consider that path (1, ..., 2) has been traversed so far in a test case. If node 3 is executed next, duas (1, (2,3), i), (5, (2,3), i) and (1, (2,3), length) might be covered provided these duas are *alive* in path (1, ..., 2). A dua is alive if it has been born

and its variable was not redefined in the path. Thus, (1, (2,3), i), (5, (2,3), i) and (1, (2,3), length) belong to the set of potentially covered duas of node 3.

Table 6 formally defines the sets of born duas (**Born**(n)), disabled duas (**Disabled**(n)), sleepy duas (**Sleepy**(n)) and potentially covered duas (**PotCovered**(n)) duas for a node $n \in N$ of a flow graph $G(N, E, s, e)$:

Table 6 – Algorithm 1 sets definitions

Born(n)	set of duas (d, u, X) or (d, (u',u), X) such that $d = n$
Disabled(n)	set of duas (d, u, X) or (d, (u',u), X) such that X is defined in n and $d \neq n$
Sleepy(n)	set of duas (d, (u',u), X) such that $u' \neq n$
PotCovered(n)	set of duas (d, u, X) or (d, (u',u), X) so that $u = n$

To determine the covered duas, BA keeps track of three working sets — the alive duas (**Alive**), the current sleepy duas (**CurSleepy**) and the covered duas (**Covered**). How these extra sets are determined is described in Algorithm 1.

```
   input  : nodes traversed during program execution;
            sets Born(n), Disabled(n), Sleepy(n) and PotCovered(n)
   output : Covered duas set
 1 Alive ← ∅;
 2 CurSleepy ← ∅;
 3 Covered ← ∅;
 4 repeat
 5 |   n ← node traversed in program execution ;
 6 |   Covered ← Covered ∪ [[Alive − CurSleepy] ∩ PotCovered(n)] ;
 7 |   Alive ← [Alive − Disabled(n)] ∪ Born(n) ;
 8 |   CurSleepy ← Sleepy(n) ;
 9 until program execution finishes;
10 return Covered
```

Algorithm 1: Bitwise dua coverage algorithm

Before the execution of the program, the three working sets are empty (lines 1 to 3). At line 5, variable n is assigned with the node traversed in the program execution. When the program execution starts, the node traversed is the start node s. The algorithm proceeds by processing the nodes traversed until the program execution finishes (line 9).

The **Alive** set contains the duas that are alive in the path so far traversed in the program execution. The duas belonging to **Alive** are those that were born in the path and were not disabled in it (line 7). Hence, the **Alive** set contains the duas enabled for coverage in the path, provided the respective c-use or p-use is met. At line 6, this meeting is verified. **Alive** is used to determine the covered (**Covered**) duas by making the intersection with the potentially covered duas (**PotCovered**) of each node. The current

sleepy duas (**CurSleepy**) is used to prevent temporally disabled p-use duas from being covered. The **CurSleepy** working set is updated at line 8 with the sleepy duas associated with the just visited node.

To illustrate how the algorithm works, consider that the path (1, 2, 3, 4) of the flow graph of Figure 3 has been traversed and processed by Algorithm 1. The contents of the working sets are: **Alive** = {(1, 4, i), (1, 5, i), (1, (2,3), i), (1, (2,6), i), (1, (3,4), i), (1, (3,5), i), (1, 4, array), (1, (3,4), array), (1, (3,5), array), (1, (2,3), length), (1, (2,6), length), (4, 6, max), (4, (3,4), max), (4, (3,5), max)}; **CurSleepy** = {all p-use duas}; **Covered** = {(1, (3,4), max), (1, 4, i), (1, (2,3), i), (1, (3,4), i), (1, 4, array), (1, (3,4), array), (1, (2,3), length)}. The next traversed node is 5, so n equals 5. Since **CurSleepy** is subtracted from **Alive** at line 6, all p-use duas are removed. The resulting set {(1, 4, i), (1, 5, i), (1, 4, array), (4, 6, max)} is subsequently intersected with **PotCovered(5)** ({(1, (3,5), max), (4, (3,5), max), (1, 5, i), (1, (3,5), i), (5, 5, i), (5, (3,5), i), (1, (3,5), array)}). Therefore, the only dua additionally covered is (1, 5, i). Therefore, the new **Covered** set is {(1, (3,4), max), (1, 4, i), **(1, 5, i)**, (1, (2,3), i), (1, (3,4), i), (1, 4, array), (1, (3,4), array), (1, (2,3), length)}.

At line 7, **Alive** is adjusted. Duas belonging to **Disabled(5)** are eliminated; that is, duas (1, 4, i), (1, 5, i), (1, (2,3), i), (1, (2,6), i), (1, (3,4), i), (1, (3,5), i) are removed. On the other hand, duas that are born at node 5 — **Born(5)** equals {(5, 4, i), (5, 5, i), (5, (2,3), i), (5, (2,6), i), (5, (3,4), i), (5, (3,5), i)} — are added to **Alive**. As a result, the value of **Alive** after processing node 5 is {(1, 4, array), (1, (3,4), array), (1, (3,5), array), (1, (2,3), length), (1, (2,6), length), (4, 6, max), (4, (3,4), max), (4, (3,5), max), (5, 4, i), (5, 5, i), (5, (2,3), i), (5, (2,6), i), (5, (3,4), i), (5, (3,5), i)}. Finally, at line 8, **CurSleepy** is set up with **Sleepy(5)**, which encompasses all p-use duas because there does not exist a p-use dua with edge (5, u).

Typically, Algorithm 1 can be implemented by inserting lines 6 to 8 at the beginning of the code associated with each node. As a result, it is not necessary to keep a list of traversed nodes. All sets utilized can be implemented as bit vectors.

In particular, the sets associated with the nodes from the flow graph (**Born(**n**)**, **Disabled(**n**)**, **Sleepy(**n**)**, and **PotCovered(**n**)**) can be encoded as constant values into the instrumented code. A global **Covered** bit vector is required for each method while local **Alive** and **CurSleepy** bit vectors are needed for each method invocation to deal with recursive calls. The size of all bit vectors is given by the number of duas of the method under analysis.

This implementation solution causes an increase in the program code; thus, the program memory is also augmented. Strategies for reducing the code growth will be discussed in Chapter 6.

The program execution might terminate without traversing an exit node $\in e$. For example, if there is an abort command in a node n such that $n \ni e$, the program will terminate its execution at n, before reaching the exit node. Nonetheless, Algorithm 1 correctly determines the covered duas until the program termination at node n (see proof of correctness in Appendix A).

4.2 Cost analysis

The analysis presented comprises RAM memory and time costs and compares three techniques for dua coverage, namely: demand-driven (DD), matrix-based (MB), and Bitwise Algorithm (BA). The DD and MB approaches were described, respectively, in Sections 3.2.2 and 3.2.3.

4.2.1 RAM Memory requirements

The memory requirements of BA as discussed above are quite limited — three bit vectors for each method. The matrix based approach (MB) [13] requires an array of (possibly) short integers to register the last definition of each variable of the method and a short integer to record the last executed node. Its more expensive data structure, though, is a matrix of short integers that occupies the number of definitions times the number of uses for each method. In addition, Santelices and Harrold use a matrix enlarged to the next highest power of two which allows shift operations to replace multiplications when cells are accessed.

The DD instrumentation [12] requires a bit vector to register the covered duas for each method, an array of (possibly) short integers to register the last definition of each variable, and a short integer to register the last executed node. However, a more expensive data structure, the *Probe Location Table* (PLT), is needed. PLT is needed to insert and remove probes since it contains the addresses for where to insert probes and also registers which probes were inserted in each node.

From the above analysis, MB and DD are clearly more demanding than BA in terms of RAM memory.

4.2.2 Time analysis

Time analysis is hampered due to the different nature of the approaches. To overcome such difficulties, our time analysis is two-fold — based on theoretical analysis and conservative simulations.

4.2.2.1 Theoretical analysis

The slowdown in an instrumented program is linear to the number of nodes traversed in a test case. Thus, we estimate the instrumentation cost per node visited. Moreover, the unit of measurement is the number of accesses to the RAM memory. Our assumption is that the memory accesses are much more expensive than the operations — except for the insertion and removal of probes, which are quite expensive.

To analyze the number of accesses of BA, firstly, consider that all sets associated with a node are not empty. In line 6 of Algorithm 1, BA requires four accesses to the RAM memory, to obtain **Alive** and **CurSleepy**; and to obtain **Covered** and return its new value back. Line 7 needs to obtain **Alive** and send it back to memory. Finally, in Line 8, one access is required to assign a new value to **CurSleepy**. Thus, BA requires seven memory accesses for each node, provided the number of duas fits in the size of the word memory; however, seven accesses is the worst case scenario. On the other hand, with **Born**(n), **Disabled**(n), **Sleepy**(n), and **PotCovered**(n) being empty, only one access is needed — to assign **Sleepy**(n) to **CurSleepy**. Since there are four sets at each node, there are 16 variations of instrumentation code in which the number of accesses varies from one to seven.

For the MB approach we estimate the number of accesses for every definition and use. Consider c-use duas $D = (d, u, X)$. At node d, one access is needed to set the ID of the last definition of X. At the use node u, the last definition of X should be obtained[1] and the matrix should be accessed to set D as covered, if this is the case. P-use duas $D = (d, (u', u), X)$ need one more access to get the last executed node. Then, for each dua, two or four accesses are necessary. However, a node may have more than one definition and one use. Moreover, in every node a memory access to set the last executed node is needed. Thus, the number of accesses per node tends to be higher than only three accesses.

Similarly, we determine the number of accesses per dua for the DD approach. At node d, one should access the last definition array to set the *new* last definition of X. Moreover, the PLT should be verified to determine if the use probe has already been inserted. At node u, the last definition of X is obtained and compared to d. For p-use duas, another access is needed to obtain the last executed node. Thus, three or four accesses are required for a dua that has not yet been covered. Like the MB approach, there may be more than one definition or use at a particular node and one access to set the value of the last visited node is also needed. Hence, the number of accesses per node should be higher than four accesses.

Additionally, Misurda et al. [12] report that a removable probe costs 25 times more than a static probe. This cost is not taken into account in the above analysis. The DD

[1] Santelices and Harrold determine at instrumentation time whether there is a single definition reaching the use to avoid this memory access.

intuition is that removable probes pay off since the instrumented program may perform as a program without instrumentation, after all duas are covered. However, this is an unlikely situation since there exist *unfeasible* duas, i.e., duas for which there is no input data that cover them [43]. To determine the unfeasible duas is in general an undecidable problem. Hence, probes to cover unfeasible duas will remain in the code.

BA will require more memory accesses if the number of duas surpasses the memory's word size. More duas implies, however, that DD and MB approaches will also have more accesses per node. Simulations were carried out to provide another way of assessing MB, DD, and BA approaches.

4.2.2.2 Simulations

Table 7 – Description of programs selected for simulation

Program	Description
Commons-Math[1]	Library of mathematics and statistics components
HSQLDB[2]	Relational database engine with in-memory and disk-based tables
JTopas[3]	Suite for parsing arbitrary text data
Scimark2[4]	Benchmark for scientific and numerical computing
Weka[5]	Collection of machine learning algorithms for data mining tasks

We utilized a program simulation framework — called *Instrumentation Strategies Simulator* (InSS) [44] — that simulates the execution of an instrumented program to assess the dua coverage approaches at run-time. The InSS engine accepts annotated graphs and duas as input. To simulate the execution of an instrumented program the InSS should receive a stream of nodes representing the program's flow of execution.

We instrumented Java bytecode programs so that whenever a node of a particular method is traversed, this information is passed to InSS. Moreover, the MB, DD, and BA approaches were implemented in Java into the InSS framework. BA implementation does not access **Born**(n), **Disabled**(n), or **PotCovered**(n) sets when their value is empty since they are useless for dua coverage determination.

The InSS simulation works as follows. Every time the first node of a new method is received from the stream of nodes, the method flow graph and its duas are loaded. An object is created representing the new method read with its probes inserted. As the traversed nodes are processed, the probes are executed performing the desired analysis. We collected only the time associated with the instrumentation code of each approach simulated.

[1] <http://commons.apache.org/proper/commons-math/> version 2.1.
[2] <http://hsqldb.org/> version 2.2.8.
[3] <http://jtopas.sourceforge.net/> version 0.8.
[4] <http://math.nist.gov/scimark2/> version 2.0.
[5] <http://www.cs.waikato.ac.nz/ml/weka/> version 3.6 revision 5178.

The simulation data is considered conservative for the following reasons: (1) The removal of probes was simulated by verifying whether a probe should be removed and withdrawn from a linked list of probes, which is comparatively less costly than altering object code. (2) For the sake of simplicity, the sets associated with each node are determined during BA simulation.

The simulations were run in a desktop with an Intel(R) Core(TM) 2 Duo CPU E4500@2.20GHz, 2,063,668 kBytes of RAM memory with Debian GNU Linux 6.0.2.

Table 7 describes the programs we have chosen to simulate the instrumented execution. Three programs — Commons-Math, Scimark2 and Weka — perform mathematical functions. JTopas parses arbitrary texts and HSQLDB is a relational database with a small memory footprint.

The programs were also chosen due to their characteristics. Weka, HSQLDB and Commons-Math are large programs and require a large number of duas. All programs have test sets with fairly long executions since the slowdown caused by an instrumentation strategy shows up more evidently in long running programs.

Table 8 – Characteristics of the selected programs

Program	Classes	Methods	LOC	duas	t-duas	cov-duas
Commons-Math	428	3,995	39,991	44,688 (81.3%)	43,233 (84.0%)	36,337
HSQLDB	439	8,365	143,532	135,529 (31.9%)	92,855 (46.6%)	43,305
JTopas	41	475	4,373	4,108 (59.9%)	3,511 (70.1%)	2,460
Scimark2	10	61	843	1,323 (53.6%)	903 (78.5%)	709
Weka	2,068	20,806	267,707	330,553 (34.0%)	169,296 (66.3%)	112,268

Table 8 presents the characteristics — number of classes, methods, lines of codes (LOC), and the total number of required duas — of the programs. Additionally, it contains the number of tracked duas (t-duas) and the number of covered duas (cov-duas) during test set execution. The number of tracked duas differs from the total of required duas (shown in column 5) to test a program. It represents only duas of methods executed during the test set. The numbers in parentheses in columns 5 and 6 represent the covered duas as a percentage of the total number of required duas and the number of tracked duas, respectively. The lines of code were measure with CLOC[7].

Table 9 shows the test set execution times and MB, DD, and BA simulation times. MB and DD simulation times are accompanied by a number in parentheses representing

[7] <http://cloc.sourceforge.net/> version 1.6.0.

Table 9 – Test set (TS) execution time and Matrix-based (MB), Demand-driven (DD), and Bitwise Algorithm (BA) simulation times

Program	TS	MB	DD	BA
Commons-Math	21.6s	218.1s (42.48%)	250.5s (63.63%)	153.1s
HSQLDB	23.6s	133.3s (98.41%)	201.5s (199.80%)	67.2s
JTopas	1.95min	116.01min (81.60%)	111.17min (74.03%)	63.88min
Scimark2	26.2s	96.3s (20.07%)	83s (3.49%)	80.2s
Weka	8.27min	328.01min (43.34%)	422.44min (84.61%)	228.83min

the percentage increase in simulation time of these strategies with respect to BA. The simulation times for Commons-Math, HSQLDB and Scimark2 are the average of ten executions whereas for the other programs they are the average of three executions, due to the longer simulation time.

Our theoretical cost analysis indicates that BA tends to require less accesses to the memory than the other approaches. The simulation data corroborates such a tendency. BA performed consistently better than the MB and DD approaches for all programs, despite the unfavorable conditions.

DD imposed a slowdown 200%, 85%, 74%, 64%, and 3% higher than BA for HSQLDB, Weka, JTopas, Commons-Math, and Scimark2, respectively. The only program for which DD has a performance comparable to BA's is Scimark2. Therefore, BA outperforms DD for all programs and significantly for four out of five programs analyzed. Regarding MB, it has a slowdown 98%, 82%, 43%, 42% and 20% higher than BA, respectively, for HSQLDB, JTopas, Weka, Commons-Math and Scimark2. Thus, BA also outperforms MB significantly for four out of five programs.

Scimark2 is the only program in which the removal of probes seems to pay off. The Scimark2 test set runs around 26s on less than 61 methods. Additionally, it contains only 903 duas to be tracked of which 709 are covered. Thus, it stays longer in each method so that the removal of probes pays off in terms of overhead reduction. Nevertheless, DD is still unable to outperform the BA. Therefore, BA and DD performances are comparable for Scimark2. Even the MB approach does not perform badly for Scimark2 since it is 20% worst than BA.

However, as the number of methods and duas grows the DD performance diminishes. JTopas and Weka have long test set executions, but also a great deal of methods and duas. As a result, methods are visited less often and a sizeable number of duas remain to be

tracked, which hampers DD performance making it significantly more costly than BA for JTopas and Weka. Commons-Math and HSQLDB have shorter test set and great amount of methods and duas. Consequently, DD performs badly in comparison to BA, especially for HSQLDB. Hence, the results suggest that BA are less affected by the increase of the number of duas than DD.

BA and MB strategies have similar characteristics since both utilize static instrumentation. However, for the analyzed programs, the BA bitwise structures and operations showed to be more efficient than the data structures and operations utilized in MB.

4.3 Final remarks

In this chapter, we presented a new approach, called the Bitwise Algorithm (BA), to track definition-use associations (duas) at run-time; this imposes limited RAM memory requirements and less slowdown. In four out of five programs analyzed, the BA performs significantly better than the previous approaches in conservative simulations.

Our next steps include instrumenting object code with the BA approach in order to investigate issues such as: (1) the BA performance in a real instrumentation; and (2) how much the code grows due to the BA instrumentation. In the next chapters, we describe the implementation of the BA-DUA (Bitwise Algorithm-powered Definition-Use Association coverage) tool, which was developed to investigate such issues.

5 Bytecode data-flow analysis

To develop a DF testing supporting tool, first it is necessary to carry out a data-flow analysis of the code. In this chapter, we present the intra-procedural DF analysis of methods compiled into Java Virtual Machine object code. We also present the ASM-DefUse, an extension of the ASM library that implements the DF analysis described in this chapter. This new library is used to compute CFG and duas directly from the object code.

5.1 The Java Virtual Machine

The Java programming language is a general-purpose, object-oriented language. Java programs are compiled according to the specification [45] into a binary format known as `class` file format and the compiled code is executed by the Java Virtual Machine (JVM) [45].

The JVM only needs to know the class file format. Despite the strong syntactic and structural constraints on its contents, any language that can be expressed in terms of a class file can be supported by JVM [45]. Examples of other languages that run on top of JVM are JRuby[1] (an implementation of the Ruby language), Jython[2] (an implementation of the Python language) and Scala[3] (a functional programming language).

A DF testing tool requires code analysis and instrumentation. In Java, this can be done at two levels — at source or object code level. Working on object code has advantages. First, the source code is not needed. Such a characteristic allows applications whose source code is not available to be supported by the tool. Another advantage is that the JVM permits the analysis and instrumentation of classes at run-time. Finally, by analyzing object code the tool can be used to support DF testing in any language compiled into JVM object code, being not restrict only to Java programs.

With these advantages, we choose to work with object code. For a better understanding, the following sections present the class file format, the JVM execution model, and the JVM instructions set in details.

5.1.1 Class file format

The class file format defines the binary representation of a single class or interface. A class or interface when compiled should be stored according to this definition to be

[1] <http://www.jruby.org/>
[2] <http://www.jython.org/>
[3] <http://www.scala-lang.org/>

executable by a JVM implementation. A class file contains *bytecode* instructions (the instructions that the JVM understands), a symbol table, and additional information.

Figure 6 presents the overall structure of the class file format. We will not provide a definition of all the fields of this binary format in this text. The interested reader may refer to the JVM specification [45] for more details.

Modifiers, name, super class, interfaces	
Constant pool (symbol table)	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code (bytecodes)

Figure 6 – Overall structure of the class file format (* means zero or more) [46].

5.1.2 JVM execution model

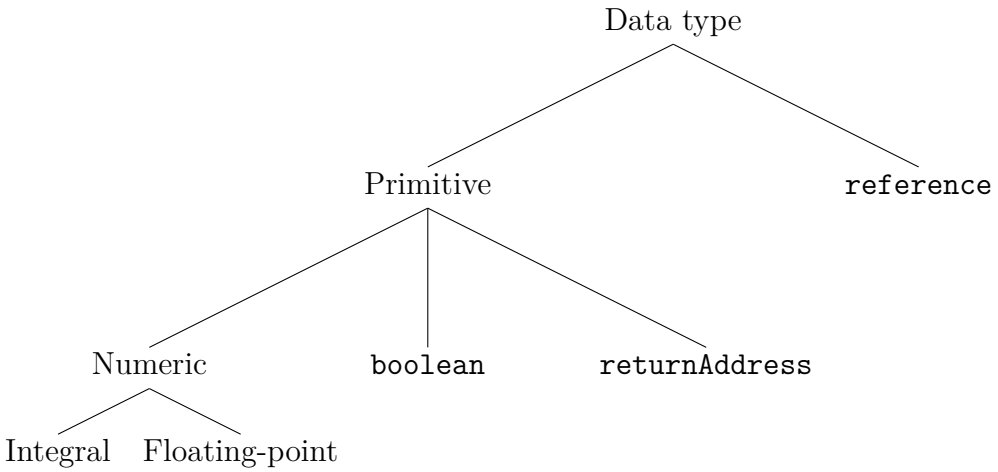


Figure 7 – Classification of Data types

The JVM operates on two kinds of data types: *primitive types* and *reference types*. There are, correspondingly, two kinds of values: *primitive values* and *reference values*.

In the JVM model, an *object* is either a dynamically allocated class instance or an array. Values that refer to objects belong to the type `reference`. A reference value is like a C pointer that refers to memory objects. Similarly to C, there may exist more than

one reference to a single object. A reference value may also be `null`, which indicates a reference to no object.

The primitive data types are divided in three types: *numeric*, `boolean`, and `returnAddress`[4]. The numeric type is split in *integral* and *floating-point* types. Figure 7 shows this classification and Table 10 describes these types.

Table 10 – Description of Data types

Data type		Values
Integral	`byte`	8-bit signed integer
	`short`	16-bit signed integer
	`int`	32-bit signed integer
	`long`	64-bit signed integer
	`char`	16-bit unsigned integer
Floating-point	`float`	32-bit single-precision floating-point number
	`double`	64-bit double-precision floating-point number
`boolean`		Encode the truth values `true` and `false`
`returnAddress`		Pointers to the opcodes of JVM instructions
`reference`		A reference to an object

JVM supports *multithreading* and a *thread* executes the code of a method. Each thread has its own `pc` (program counter) register that contains the address of the bytecode being currently executed and also has a private *stack* created along with the thread that stores *frames*.

A *frame* represents a method invocation. It holds local variables, stores partial results, returns values and dispatches exceptions. For a given thread, only the frame at the top of the stack is active at any point. This frame is referred to as the *current frame*, and its method is known as the *current method*. The class in which the current method is defined is the *current class* [45].

When a method is invoked, a new frame is created and pushed on the stack. The control then transfers to the new method. When the method returns, either normally or abruptly (because of an uncaught exception), the current frame passes back the result of its method invocation, if any, to the previous frame. The current frame is then popped from the stack. The previous frame becomes the current one and control transfers back to the calling method.

Each frame has its own *local variables* array. The size of the local variable array is fixed and is provided along with the code for the method. A single local variable can hold values of any type, except `long` and `double` values. For these types, two consecutive local variables are required. Local variables are addressed by the index in the array (starting from the index zero). Values of type `long` or `double` utilize the lesser index value.

[4] The `returnAddress` type is used by some deprecated instructions and is not covered here.

When a class method (static method) is invoked, all parameters are passed through a local variable array. The first parameter is passed through the local variable 0 and subsequent parameters are passed through consecutive local variables. For an object method invocation, the same principle is applied, but the local variable 0 is used to pass the reference to the class instance whose method is being invoked (`this` in the Java programming language).

A frame also has its own *operand stack*. When a frame is created its operand stack is empty. The JVM has some instructions that are used for *data transfer*. For instance, there are instructions to push values (constants or values from local variables and fields) onto the operand stack. There are instructions to do the reverse too, i.e., pop a value from the operand stack and store in a local variable or in a field. Other instructions take operands (as arguments) from the operand stack, perform some computation, and push the result back onto the operand stack. The operand stack has a maximum depth that is fixed and is determined at compile-time. Values of type `long` or `double` contribute two units to the depth of the operand stack. All other value types contribute one unit.

Finally, the JVM has a run-time data area, known as *heap*, where all class instances and arrays are allocated. The heap is shared among all threads and its memory is reclaimed by the *garbage collector*. Another data area that is shared among all threads is the *method area*. It stores class structures and the code for methods.

5.1.3 Bytecode instructions

A bytecode instruction is a one-byte *opcode* that defines an operation to be performed, followed by zero or more *operands* that define the arguments for the operation. The opcode is identified by a mnemonic symbol and the arguments are a fixed number of static values[5].

For instance, Oracle's JVM implements the unary increment operation of integer local variables with a bytecode instruction whose opcode is identified by the mnemonic `iinc`. This instruction takes two arguments: a local variable index (its first operand) and a one-byte signed value (its second operand). The result is the increment of the local variable determined by the first operand by the value defined in the second operand.

As stated in the previous section, some instructions utilize operands from the operand stack. For example, the `iadd` instruction adds two `int` values. Both the `int` values are popped from the operand stack and added, and their sum is pushed back onto the operand stack. To prevent confusion, we call these values taken from the operand stack *run-time arguments* and the operands defined by the bytecode instruction *static arguments*.

[5] The number and size of the operands are determined by the opcode.

A *definition* occurs by instructions that perform data transfer. For instance, the instruction `istore` takes one static argument i (a local variable index of the current frame). This instruction also needs a run-time argument x that must be of type `int`. The `istore` instructions set the value of the local variable i by x. In this case we say that a *definition* of the local variable addressed by i occurred. The same principle is used for instance and class fields.

There are conditional and unconditional transfer bytecode instructions. Both transfer instructions takes a *branch offset* static argument. This argument is an offset from the address of the transfer instruction opcode to the *target* bytecode instruction. For simplicity, the target of transfer instructions is described in this work by an index instead of a branch offset. We define the code of a method as a finite sequence of valid bytecode instructions $(b_1, \ldots, b_n)$ of length n, where each instruction b_i is identified by its index i in the sequence. These are the same definitions of Section 2.3. Indeed, control-flow concepts (e.g., node, edge, CFG) described in Section 2.3 can be obtained from the bytecode instructions of a method.

Most of the instructions in the JVM instruction set encode type information regarding the operations they perform. For instance, the `iadd` instruction adds two values from the operand stack that must be of type `int`. The `fadd` does the same with a `float` value. Annex A lists all bytecode instructions.

5.2 Bytecode analysis

5.2.1 Control-flow analysis of bytecodes

Consider the Java method in Figure 2a that determines the maximum element in an array of `int`. Table 11 shows the bytecode instructions generated by a Java compiler from the method. The bytecode instruction sequence in Table 11 is divided in disjoint blocks. The first column in the table identifies the basic block. Each basic block has a corresponding node in CFG of Figure 2b. The block id in Table 11 is the same as that the corresponding node in Figure 2b. The second and third columns list the index and the bytecode instructions, respectively.

For instance, the first instruction from node 2 (`iload_3`) pushes the value from local variable 3 (the variable named i in the source code) onto the operand stack. The instruction `iload_2` pushes the value from local variable 2 (the variable named *length* in the source code) onto the operand stack. The last instruction of this node (`if_icmpge`) pops the two values, executes a predicate to test if the second popped value (variable i) is greater than or equal to the first popped value (variable *length*). The predicate checks the while condition ($i < length$).

Table 11 – Bytecode instructions for method max of Figure 2a

Id	Index	Bytecode	Operand stack	Def	Uses
1	1	iconst_0	`int`		
	2	istore_3		l_3	
	3	aload_1	l_1		
	4	iinc 3, 1	l_1	l_3	l_3
	5	iload_3	l_1, l_3		
	6	iaload	$l_1[l_3]$		
	7	istore_4		l_4	l_1, l_3
2	8	iload_3	l_3		
	9	iload_2	l_3, l_2		
	10	**if_icmpge** 25			l_2, l_3
3	11	aload_1	l_1		
	12	iload_3	l_1, l_3		
	13	iaload	$l_1[l_3]$		
	14	iload_4	$l_1[l_3]$, l_4		
	15	**if_icmple** 20			l_1, l_3, l_4
4	16	aload_1	l_1		
	17	iload_3	l_1, l_3		
	18	iaload	$l_1[l_3]$		
	19	istore_4		l_4	l_1, l_3
5	20	iload_3	l_3		
	21	iconst_1	l_3, `int`		
	22	iadd	(l_3,`int`)		
	23	istore_3		l_3	l_3
	24	goto 8			
6	25	iload_4	l_4		
	26	ireturn			l_4

From the method code, ASM-DefUse builds an in-memory data structure to represent a CFG. This process is carried out in two steps. In the first step, a data structure is created to represent an instruction level flow graph. Every node in this graph represents an instruction and the edges represent the possible flow of control between these instructions. In the second step, a new data-structure is created to represent basic blocks.

Consider a method bytecode instruction sequence $(b_1, \ldots, b_n)$ of length n. Internally, the instruction level flow graph keeps three arrays of length n. The `instruction` array represents the sequence of bytecodes, so that the value of the array at index i refers to bytecode instruction b_i. The other two arrays represent the flow of control and are used to implement an adjacency list — an array to `successors` and another to `predecessors`. Similar to the instruction array, successors and predecessors from a bytecode instruction b_i are addressed by the index of i. The values in the adjacency lists are integer indices such that if b_i executionally precedes b_j, then `successors[i]` contains j. All arrays are filled with the help of the ASM library (see Section 5.2.3).

The basic blocks are represented by other two arrays: an integer array of length n (`leaders`) is used to indicate which basic block an instruction belongs to. For example, the thirteenth instruction of Table 11 belongs to basic block 3. In this case, `leaders[13]` equals 3. Finally, the last array (`basicblock`) initially has size n, but after the calculation

of the basic blocks it is shrinked to the number of basic blocks. This array keeps a set with the indices of instructions that belong to each basic block (a set for each entry in the array). For example, given the instruction sequence in Table 11, then `basicblock[2]` equals {8, 9, 10}.

5.2.2 Data-flow analysis of bytecodes

The difficulty in analyzing bytecode programs resides, though, in capturing DF information; that is, *uses* and *definitions* of variables. We start off by defining *uses* in the bytecode context. Three kinds of uses, both with respect to the operand stack, are considered.

Push use: this use occurs when an instruction pushes some value onto the operand stack. For instance, `iconst_1` pushes the constant `int` value 1 onto the operand stack. The instruction has a use of the constant value. Another example involving local variables is the instruction `iload_0`. This instruction loads the local variable 0 onto the operand stack. In this case, there is a use of this local variable.

Pop use: this use occurs when an instruction pops a value from the operand stack. For instance, `iadd` takes two values from the operand stack and pushes the result back. This instruction has a use of both popped values.

Definitive use: this is the same as a *pop use* but these instructions do not push any value back on the operand stack.

For *definitions* we need to define what is being assigned. Three types of variables can be assigned: *local variable*, *object field*, and *class field*. Bytecode assignment instructions such as `istore`, `putfield`, and `putstatic` assign a value to local variable, object field, and class field, respectively. They perform definitive use of values in the operand stack. Assignment instructions are always associated with a definition of a variable and uses of values. For instance, `istore_0` uses the value on the top of the operand stack and defines the local variable 0. All bytecode instructions perform a push use or a pop use.

5.2.3 Collecting control- and data-flow information

Rather than working directly on the binary class file, we use the ASM library for analyzing and instrumenting programs compiled into bytecodes. ASM provides a high level abstraction of a bytecode program. It represents a program as a tree in such a way that the methods of its API (Application Programming Interface) allow forward DF analysis to be implemented. The implementation is divided in two parts: a fixed part, defined by

the `Analyzer` and `Frame` classes; and a variable part, defined by implementations of the `Interpreter` abstract class and `Value` interface.

A `Value` object represents a value on the operand stack or in the local variable array. A `Interpreter` works like a semantic bytecode interpreter of `Value`s. The `Analyzer` depends on the `Interpreter`; the former carries out all operations that manipulates the operand stack and transfers values from/to the local variable array. The `Interpreter` implements the semantics of the operation.

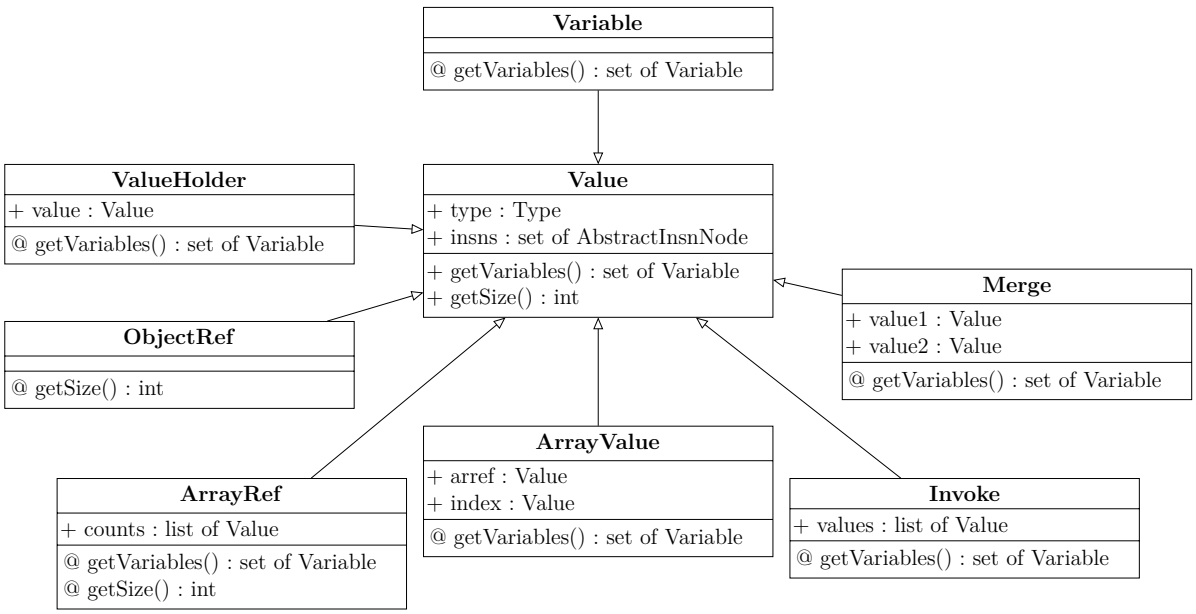


Figure 8 – Value inheritance

ASM-DefUse extends the ASM `Value` class with their own `Value` class. This class represents a general value on the operand stack or in the local variable array. Figure 8 presents a UML class diagram of this class and its sub-classes. As said before, a `Value` represents a general value; it has a field `type` to indicate the data type. This class also defines two methods. The method `getSize()` uses the field `type` to determine whether this value needs one or two local variables to be stored in the local variable array. This method is also used to indicate whether this value contributes two units to the depth of the operand stack. The other method `getVariables()` returns a set of `Variable`s that generates the value. The default implementation of this method returns a empty set. Note that `Variable` is also a sub-class of `Value`.

`Variable` is an abstract class that represents a value loaded from one of the variables (local variable, object field and class field). Figure 9 shows the UML class diagram of this class and its sub-classes. These classes implement methods `hashCode()` and `equals()` so their objects can easily be compared. The @ sign in the class diagram indicates that the method is overwritten. Also, `Variable` overwrites the behavior of method `getVariable()` so that now it returns a set with a single object representing itself. The only exception

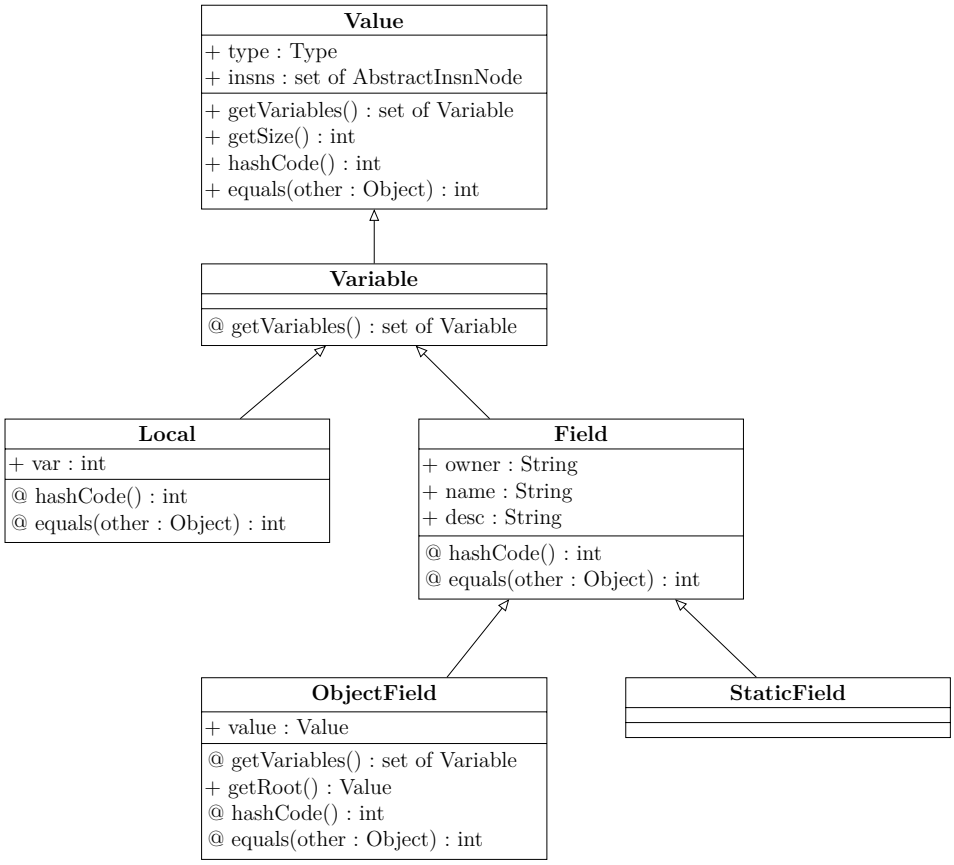


Figure 9 – Variable inheritance

is class `ObjectField`. This class has a field of type `Value` that represents a reference to an object. The method `getVariables()` of `ObjectField` returns a set with all variables from its field `value` (i.e., `value.getVariables()`).

In addition, the set contains itself, if and only if, `value` is also an instance of variable; and if `value` is instance of `ObjectField`, the same rule applies for their values.

5.2.4 A complete example

To illustrate, let us consider the program presented in Figure 2a and its bytecode sequence in Table 11. The fourth column in this table lists the operand stack immediately after the execution of the respective instruction. The fifth and sixth columns show definitions and definitive uses, respectively.

Let us consider node 3. The first instruction loads the local variable 1 onto the operand stack. The ASM-DefUse interpreter create a `Value` instance that represents this value. In this case, a `Local` instance is created with the field `var` = 1 (represented by l_1). The second instruction of this node pushes l_3 on the operand stack. So at this moment, we have two `Local` values (l_1 and l_3) instances in the operand stack.

The next instruction loads the value from the array onto the operand stack. This

instruction takes two run-time arguments, the array reference (l_1) and the index (l_3) where the value should be loaded. We represent this value by an instance of the `ArrayValue` class. This class has two fields — one for each argument taken from the operand stack. The next instruction pushes l_4 on the operand stack. So at this point, we have two `Values` ($l_1[l_3]$ and l_4) in the operand stack ($l_1[l_3]$ represent an instance of `ArrayValue` and l_4 an instance of `Local`). Finally, the last instruction of node 3 (`if_icmple`) takes two run-time arguments and executes a predicate.

The algorithm to compute duas is the same used by ASSET. For each instruction d that has a definition of X, a depth-first search is performed. The search visits every instruction u which is syntactically reachable from d by some definition-clear path. If instruction u has a definitive use of X then a dua is created. When u is a conditional transfer instructions, p-use duas are created.

5.3 Final remarks

In this chapter, we presented the Java Virtual Machine (JVM) and the DF analysis in its context. We introduced some new concepts of variable definition and uses in bytecode programs. Finally we presented a new library, namely ASM-DefUse, that implements these analysis. We implemented ASM-DefUse especially for use in BA-DUA (Bitwise Algorithm-powered Definition-Use Association coverage) tool but it is a general-purpose library for DF analysis of bytecodes. ASM-DefUse is available for download and use at <http://github.com/saeg/asm-defuse/>. In the next chapter, we describe the implementation of the BA-DUA tool.

6 BA-DUA tool

The Bitwise Algorithm (BA) for dua monitoring is realized in the BA-DUA tool, which is described in this chapter. We also discuss strategies for reducing the code growth and for optimizing the execution at run-time.

6.1 BA-DUA – BA-powered Definition-Use Association coverage – tool

BA-DUA is implemented as two Java programs. The *instrumenter* and the *reporter*. The former is responsible for instrumenting files in the class file format in order to track duas at run-time. The reporter is in charge of analyzing which duas were (or were not) covered during the execution of a test set, i.e., it creates a dua coverage report of a test set.

BA-DUA can instrument classes in two different ways:

- *on-the-fly* instrumentation modifies classes as they are loaded by the JVM. In this case, the BA-DUA instrumenter agent should be included by specifying an option in the command-line;
- *off-line* instrumentation changes specific classes before the program starts. The instrumented classes should be in the classpath so that at run-time they are loaded instead of the original classes.

The instrumenter works as follows. For each method, a flow graph is built based in the sequence of the method bytecodes. Initially, every instruction is a node and the edges represent the possible flow of control between the instructions. This graph is then transformed into a new graph where each node is a sequence of instructions. All instructions in these nodes are always executed in sequence (except when exceptions occurs). Finally, for each instruction, we compute the variables (local variables and fields) that are defined/used by the instruction. This information will generate the sets of definitions/uses of each node. Variable definitions, variable uses and duas are then computed using the ASM-DefUse tool as described in Chapter 5.

Given the annotated flow graph of a program method and its set of duas, the instrumenter inserts BA code into the program bytecode sequence. If the method does not require any dua (e.g., straight line methods), no instrumentation code is inserted. Otherwise, the instrumenter decides how to insert additional BA code into program. Our

design rationale was to avoid any unnecessary method call since we assume that method calls are more costly than inline instrumentation code.

If a method has up to 32 duas, we use integers to track them. Otherwise we use longs. Each bit in the primitive type corresponds to a dua. We do not need to worry about the size of primitive type since JVM specifies that `int` and `long` are always 32-bits and 64-bits wide, respectively.

Alive, **CurSleepy** and **Covered** working sets are translated into integer or long local variables. Methods with no duas do not need any instrumentation code, so no local variables are needed in this case. If a methods has up to 32 duas, `int` variables are used. All other methods use `long` variables. If a method has more than 64 duas, no primitive type can hold them. In this case, our solution is to use more local variables. Our experimental data suggests that these methods constitute a small part of a program (see Table 15). BA-DUA always utilizes `long` variables when the number of duas of the method exceeds 64.

Each one of these local variables belongs to a *window*. For instance, if a method has 150 duas, then the instrumentation will allocate three windows: $(dua_0, dua_1, ..., dua_{63})$ belongs to $window_0$, $(dua_{64}, dua_{65}, ..., dua_{127})$ belongs to $window_1$, and so on. In that sense, to hold **Alive**, **CurSleepy** and **Covered**, methods with up to 32 duas need only one associated window and need three `int` local variables. Methods with up to 64 duas also need only one associated window but need three `long` local variables. Methods with more than 64 duas need $\lceil |D|/64 \rceil$ associated windows, where D is the set of duas of a method. These methods need three `long` local variables for each window.

For each node n, we compute the sets **Born**(n), **Disabled**(n), **PotCovered**(n) and **Sleepy**(n) at instrumentation time. These sets are encoded as constants of type `int` or `long`, depending on the number of duas. Again, each one of these constants are encoded for each window.

```
...
// 1. Update covered
covered = covered | ((alive & ~cursleepy) & POT_COVERED_N);
// 2. Update alive
alive = (alive & ~DISABLED_N) | BORN_N;
// 3. Update sleepy
cursleepy = SLEEPY_N;
...
```

Figure 10 – BA probe for a node n

To translate Algorithm 1 into bytecodes we insert the operations that update **Covered**, **Alive** and **CurSleepy** at the beginning of each node of the flow graph. The code is inserted just before the first instruction of the node. The inserted code only collects coverage information without changing the method behavior.

Suppose we are instrumenting node n of a method with up to 32 duas. The code in Figure 10 describes how a BA *probe* looks like in Java source code. `covered`, `alive` and `cursleepy` are local variables of type `int` and represent **Covered**, **Alive** and **CurSleepy** working sets of Algorithm 1. `POT_COVERED_N`, `DISABLED_N`, `BORN_N` and `SLEEPY_N` are constants of type `int` and represent constant sets **PotCovered**(n), **Disabled**(n), **Born**(n) and **Sleepy**(n) of Algorithm 1.

The union and intersection of two bit vectors (local variables) are implemented by the bitwise inclusive `OR` and `AND` operators, respectively. The subtraction operation is implemented by negating the subtrahend and making a bitwise `AND`. If a method has more than one associated window, then the code in Figure 10 is replicated, but by using local variables and constants of each respective window. The instrumentation code associated with a window is called a *probe*.

Table 12 – BA probe bytecode instructions generated by `javac`

Bytecode	Operand stack
iload `covered`	`covered`
iload `alive`	`covered`, `alive`
iload `cursleepy`	`covered`, `alive`, `cursleepy`
iconst_m1	`covered`, `alive`, `cursleepy`, -1
ixor	`covered`, `alive`, `~cursleepy`
iand	`covered`, (`alive` & `~cursleepy`)
push `POT_COVERED_N`	`covered`, (`alive` & `~cursleepy`), `POT_COVERED_N`
iand	`covered`, ((`alive` & `~cursleepy`) & `POT_COVERED_N`)
ior	(`covered` | ((`alive` & `~cursleepy`) & `POT_COVERED_N`))
istore `covered`	
iload `alive`	`alive`
push `~DISABLED_N`	`alive`, `~DISABLED_N`
iand	(`alive` & `~DISABLED_N`)
push `BORN_N`	(`alive` & `~DISABLED_N`), `BORN_N`
ior	((`alive` & `~DISABLED_N`) | `BORN_N`)
istore `alive`	
push `SLEEPY_N`	`SLEEPY_N`
istore `cursleepy`	

To convert the statements from a BA probe to bytecode instructions is straightforward. One can write the statements in Figure 10 as a valid method, compile it with a Java compiler (`javac`) and use a Java decompiler (`javap`) to see the generated code, i.e., the BA probe of a node n for a single window. Using this approach, Table 12 shows the bytecode instructions of Figure 10. The first column shows the bytecode instructions, the second column lists the state of the operand stack, but presenting the operation as values.

Nevertheless, the bytecode instructions generated by BA-DUA for a BA probe is different from that generated by `javac`. BA-DUA optimizes the sequence of instructions of a BA probe to reduce the maximum operand stack depth needed to execute the probe. The operand stack for a BA probe generated by BA-DUA has at most two values at a time. Table 13 shows the probe generated by BA-DUA.

Table 13 – BA probe bytecode instructions generated by BA-DUA

Bytecode	Operand stack
iload `cursleepy`	`cursleepy`
iconst_m1	`cursleepy`, -1
ixor	`~cursleepy`
iload `alive`	`~cursleepy`, `alive`
iand	(`~cursleepy` & `alive`)
push `POT_COVERED_N`	(`~cursleepy` & `alive`), `POT_COVERED_N`
iand	((`~cursleepy` & `alive`) & `POT_COVERED_N`)
iload `covered`	((`~cursleepy` & `alive`) & `POT_COVERED_N`), `covered`
ior	(((`~cursleepy` & `alive`) & `POT_COVERED_N`) | `covered`)
istore `covered`	
push `~DISABLED_N`	`~DISABLED_N`
iload `alive`	`~DISABLED_N`, `alive`
iand	(`~DISABLED_N` & `alive`)
push `BORN_N`	(`~DISABLED_N` & `alive`), `BORN_N`
ior	((`~DISABLED_N` & `alive`) | `BORN_N`)
istore `alive`	
push `SLEEPY_N`	`SLEEPY_N`
istore `cursleepy`	

We note that a BA probe needs 18 instructions. In the bytecode listings of Tables 12 and 13, `covered`, `alive` and `cursleepy` should be replaced by the index of the corresponding local variable. Also, the instruction `push X` is not a valid bytecode instruction. These instructions should be replaced by valid bytecode instructions that push an `int` value on the operand stack (e.g., `icont_1`, `bipush`, `sipush` or `ldc`). The instruction used will depend on the value that will be pushed. Then `POT_COVERED_N`, `DISABLED_N`, `BORN_N` and `SLEEPY_N` should be replaced by a constant value as a immediate argument for the `push` instruction or as a index corresponding to the value in the method symbol table.

Seven out of eighteen instructions from a BA probe generated by BA-DUA are 1-byte instructions (`iconst_m1`, `ixor`, `iand` and `ior`). `iload X` and `istore X` instructions generated by BA-DUA have at least two bytes and at most four bytes, depending on the index of the variable. Finally, the instructions represented by `push X` have at least one byte and at most three bytes. This implies that the minimum size of the BA probe generated by BA-DUA is 25 bytes and the maximum size is 47 bytes.

At the beginning of the method code, instructions are added to initialize the new local variables of each window with the empty value (i.e., 0). Because `covered` is a local variable in each window, it is not sensitive to recursive calls and threads. However, its scope is limited to the method call. To keep the coverage between method calls, each class has a reference to a global array of covered duas. BA-DUA implements the global array as one array of longs (`long[]`) per class. A method can be associated with more than an entry (for instance, if a method have more than 64 duas).

Just before a `return` or a `throw` command, instructions are added to update the correspondent entries in the global array of covered duas. As mentioned before, the local `covered` variable is not thread sensitive; however, the global array entries are. Therefore,

race conditions may occur when entries of the global array are updated. As a result, the coverage obtained can be slightly lower than the real coverage for programs with multiple threads. Strategies for dealing with race conditions are beyond the scope of this work.

6.2 The BA-DUA instrumentation: an example

To exemplify how BA-DUA works, let us consider the simple example of Figure 11a. For simplicity sake, the BA-DUA bytecode instrumentation is showed by changing the source code. Thus, the description is shorter and clearer.

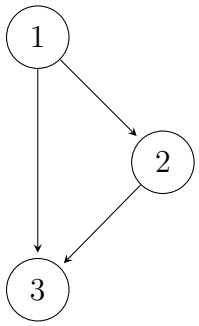
Figure 11 – A simple example of instrumentation with BA-DUA

```
public class Next {

   /*1*/ public static int odd(int x) {
   /*1*/     if (x % 2 != 0) {
   /*2*/         x++;
   /*2*/     }
   /*3*/     x++;
   /*3*/     return x;
   /*3*/ }

}
```

(a) Example program



(b) CFG of the example program

Consider the Java class described in Figure 11a with a method that determines the next odd number from a value received by parameter. The CFG from this method is presented in Figure 11b. The number inserted inside the comments refers to the node each statement is associated with. All nodes have definitions of variable x. Edges (1,2) and (1,3) have p-uses of variable x due to the predicate associated with node 1.

This method has 5 duas that are listed here: (1, 2, x), (1, 3, x), (1, (1,2), x), (1, (1,3), x) and (2, 3, x). Suppose these duas are associated with bits in the same order. Thus, (2, 3, x) is associated with the most significant bit. Figure 12 shows the instrumented source code of the program of Figure 11a.

The first thing to note in the instrumented version of the example program is the field `$data`. This is the global array of covered duas. This field is a reference to an array of longs. It starts with the `null` value and is only initialized the first time it is used. The instrumented code accesses this field through method `$getData()`. This method checks if `$data` is already initialized; in this case, it simply returns the reference to the array of longs. Otherwise, if `$data` is not initialized yet (if its reference is to `null`), then the BA-DUA method that creates a new array instance is invoked. The second argument for this method is the size of the array to be alocated. This value is determined at instrumentation time after analyzing all methods. Internally, BA-DUA keeps a hash map associating the name of the class with the array of longs.

```
public class Next {

    private static long[] $data;

    private static final long[] $getData() {
        if ($data == null) {
            $data = BADUA.getData("Next", 1);
        }
        return $data;
    }

    public static int odd(int x) {
        // setup probe
        int covered = 0;
        int alive = 0;
        int cursleepy = 0;
        // node 1 probe
        covered |= (alive & ~cursleepy) & 0;
        alive = (alive & ~0b10000) | 0b01111;
        cursleepy = 0;
        if (x % 2 != 0) {
            // node 2 probe
            covered |= (alive & ~cursleepy) & 0b00101;
            alive = (alive & ~0b01111) | 0b10000;
            cursleepy = 0b01100;
            x++;
        }
        // node 3 probe
        covered |= (alive & ~cursleepy) & 0b11010;
        alive = (alive & ~0b11111) | 0;
        cursleepy = 0b01100;
        x++;
        // update coverage
        $getData()[0] |= covered;
        return x;
    }

}
```

Figure 12 – Example program instrumented

BA-DUA manages the global array of covered duas at run-time. Thus, it can be accessed anytime. Additionally, BA-DUA registers a shutdown hook in JVM so that when JVM terminates the program execution BA-DUA traverses the hash map writing its contents to the disk. As a result, coverage data for future analysis is saved.

Another thing that should be noted is that at the beginning of the method, code is added to create and initialize the local variables representing the working sets with the empty value (i.e., 0). This is the *setup probe*. The BA probes — those in charge of dua monitoring — were described in the previous section. Finally, the last thing to note is the *update probe*. Since `covered` is a local variable, it should be exported to the global array. The code added by the update probe carries out this task. It takes the previous value from the global array, updates its value and stores the updated value back.

Our instrumentation approach has some disadvantages. Firstly, since we are adding

instrumentation code at the beginning of the node, we are being optimistic about the program execution. A dua might be marked as covered even though its use is not reached. For instance, a dua is covered in a node provided its use is reached. However, the node execution may be interrupted by an exception before reaching the use. In this case, the path between the definition and the use was not actually traversed. If the current method catches the exception and then terminates normally, there may exist duas marked as covered that in fact were not exercised. In this case, the dua coverage will be slightly overestimated.

Another disadvantage is that duas covered by methods that terminate abruptly do not contribute to the dua coverage. This is so because the global array is only updated at the normal method exit. Therefore, the dua coverage will be underestimated. We intend to address this limitation in future versions of BA-DUA.

6.3 Optimizing BA probes

Since all sets of node n are static, the BA code can be optimized at instrumentation time. Some of these optimizations are supposed to happen when the JVM Just-In-Time compiler optimizes the code. But optimizations to reduce the size of the probe are important since the maximum size of a method code in the class file format is 64 kBytes and the inclusion of BA probes may exceed that limit. All the following optimizations were implemented in BA-DUA.

Let us consider the probe from Figure 10 (i.e., the single window BA code for a node n). When `POT_COVERED_N = 0`, statement 1 can be removed. In this case there are no duas (d, u, X) or (d, (u',u), X) such that $u = n$. Thus, no dua may be covered at node n, which makes this statement unnecessary. Another way to prove it is showing that `((alive & ~cursleepy) & POT_COVERED_N)` returns 0 and the value of `covered` will not change.

When `BORN_N = 0` and `DISABLED_N = 0`, statement 2 is removed. When only `BORN_N = 0` statement 2 can be rewritten as `alive = alive & ~DISABLED_N` . Finally, statement 2 can be rewritten as `alive = alive | BORN_N` when only `DISABLED_N = 0`. In statement 2, the value of `~DISABLED_N` is determined at instrumentation time.

The statement 3 is rewritten as `cursleepy = ~SLEEPY_N`, i.e., `cursleepy` now stores all duas enabled to be covered at the next node. This update save two instructions needed to negate `cursleepy` in statement 1. Note that `~SLEEPY_N` can be determined at instrumentation time. With this update, the setup probe should initialize `cursleepy` with all bits on.

At the start node s, the statement 1 is removed by the rule of `POT_COVERED_N =`

0. In addition, we just need to update `cursleepy` when **Sleepy(s)** is not empty. Note that the setup probe already sets all variables to empty. Furthermore, since at the start node `alive` is empty, we do not need to subtract **Disabled(s)**. Similarly, exit nodes only need to update `covered`. Updating `alive` and `cursleepy` is unnecessary since no node might be executed after the exit node.

Finally, when a node has only one predecessor, or, when `POT_COVERED_N` does not keep any p-use duas, statement 1 does not need `cursleepy`. In this case, statement 1 can be rewritten as `covered = covered | (alive & POT_COVERED_N)`. Figure 13 presents the same instrumented program of Figure 12, but with optimized probes.

```
public static int odd(int x) {
    int covered = 0;
    int alive = 0;
    int cursleepy = -1;
    alive = 0b01111;
    if (x % 2 != 0) {
        covered |= alive & 0b00101;
        alive = (alive & 0b10000) | 0b10000;
        cursleepy = 0b10011;
        x++;
    }
    x++;
    $getData()[0] |= covered | (alive & cursleepy) & 0b11010;
    return x;
}
```

Figure 13 – Example program instrumented (optimized)

6.4 Final remarks

In this chapter, we presented the implementation of BA-DUA (BA-powered Definition-Use Association coverage) tool. We also discussed the strategies for reducing the code growth by optimizing the probe size. BA-DUA is available for download and use at <http://github.com/saeg/ba-dua/>. In the next chapter, an experiment is carried out to evaluate the penalties imposed by BA instrumentation with BA-DUA.

7 Evaluation

In this chapter, we present an evaluation of the BA in the BA-DUA tool implementation. Our evaluation comprehends two types of penalties imposed by instrumenting programs with BA: execution overhead, given by the extra time needed to execute the instrumentation code with respect to an uninstrumented program; and memory overhead, represented by the growth of the object code due to the insertion of BA code.

7.1 Experimenting with BA-DUA and JaCoCo

An experiment to evaluate the penalties imposed by BA was carried out. The experiment was designed to answer the following research questions:

RQ1: How do the penalties imposed by BA impact on an instrumented program?

RQ2: How much more expensive is dua monitoring using the BA approach in comparison with edge monitoring?

Two kinds of penalties were assessed: execution overhead, given by the extra time needed to execute the instrumentation code added; and memory overhead, represented by the growth of the object code due to the insertion of instrumentation code.

BA-DUA was used to assess dua monitoring whereas JaCoCo was utilized to assess edge monitoring. This tool was chosen because it has been used to obtain control-flow (CF) — instruction and edge — coverage of large systems. Both BA-DUA and JaCoCo require limited extra memory at run-time; thus, the program growth provides an estimate of the extra memory needed to run both instrumentation approaches. Next, we describe the evaluation and results.

7.2 Subject programs

Table 14 contains the description of the programs used in our evaluation. We utilized several criteria to select them. The first criterion was to select programs that were previously used in DF testing instrumentation papers. The idea was to allow a rough comparison between instrumentation techniques. These programs are indicated in Table 14 by a star ($\star$). With this same criterion, we added those programs used in the simulations contained in [14] and also in Chapter 4. They are referred to by a diamond ($\diamond$).

Table 14 – Description of the programs selected for evaluation

Program	Description
Commons-Lang[1]	Library with utilities classes for Java
Commons-Math 2.1 (◇) and 3.2	Library of mathematics and statistics components
HSQLDB (◇)	Relational database engine with in-memory and disk-based tables
JFreeChart[2]	Library to display professional quality charts in applications
JTopas (⋆,◇)	Suite for parsing arbitrary text data
Scimark2 (⋆,◇)	Benchmark for scientific and numerical computing
Weka r5178 (◇) and r10042	Collection of machine learning algorithms for data mining tasks
XML-Security[3] (⋆)	Library to implement XML signature and encryption standards

The second selection criterion was based on the characteristics of the systems. The goal was to have a diverse selection of programs. Five programs — Commons-Math 2.1 and 3.2, Scimark2, and Weka r5178 and r10042 — perform mathematical calculations. JTopas parses arbitrary texts, XML-Security parses and manipulates XML files, and HSQLDB is a relational database with a small memory footprint. Thus, these three programs perform data manipulation tasks. Finally, JFreeChart is a graphics library and Commons-Lang is a library that performs tasks related to string manipulation, concurrency, creation and serialization of objects, among others. All programs are either open-source or publicly available.

The third criterion is related to the size of the program. To assess the scalability of BA, we defined three ranges of systems: small, medium, and large. Small systems are those up to 5,000 LOC; medium size systems vary from 5 KLOC up to 50 KLOC; and large systems were those with more than 50 KLOC. These ranges are related to the number of engineers working in the system. A small program can be easily coded by a single engineer; a medium size program can be coded by a single engineer in the bottom of the range (5 KLOC), but more help will be needed when its size reaches the top of the range (50 KLOC). Large systems need a group of engineers.

Table 15 describes the characteristics of the selected programs. It shows the number of lines of code (LOC), the number of classes and methods, the number of methods with more than 64 duas, with more than 32 up to 64 duas, with up to 32 duas, and with zero duas; it also contains the total number of duas and of edges. The percentage numbers in

[1] <http://commons.apache.org/proper/commons-lang/> version 3.1.
[2] <http://www.jfree.org/jfreechart/> version 1.0.16.
[3] <http://santuario.apache.org/> version 1.5.5.

Table 15 – Characteristics of the selected programs

Program	LOC	Classes	Methods	Methods w. #duas>64	Methods w. 64≥#duas>32	Methods w. 32≥#dua>0	Methods w. #duas=0	Duas	Edges
Commons-Lang	19,499	143 (99.30%)	2,281 (93.03%)	36 1.58%	128 5.61%	901 39.50%	1,216 53.31%	20,901 (87.67%)	7,395 (91.64%)
Commons-Math 2.1	39,991	428 (99.53%)	3,995 (86.43%)	130 3.25%	194 4.86%	1,096 27.43%	2,575 64.46%	41,296 (85.85%)	8,745 (85.93%)
Commons-Math 3.2	81,792	845 (95.03%)	6,886 (85.42%)	285 4.14%	358 5.20%	1,709 24.82%	4,534 65.84%	88,369 (77.62%)	18,576 (84.77%)
HSQLDB	143,532	439 (66.29%)	8,365 (48.46%)	432 5.16%	626 7.48%	2,997 35.83%	4,310 51.52%	128,854 (37.07%)	34,722 (34.59%)
JFreeChart	93,322	520 (82.88%)	8,313 (60.60%)	309 3.72%	361 4.34%	1,943 23.37%	5,700 68.57%	84,678 (44.50%)	21,165 (45.10%)
JTopas	4,373	41 (78.05%)	475 (75.37%)	13 2.74%	19 4.00%	147 30.95%	296 62.32%	3,951 (65.65%)	1,233 (61.80%)
Scimark2	843	10 (90.00%)	61 (62.30%)	4 6.56%	6 9.84%	26 42.62%	25 40.98%	1,186 (59.11%)	218 (59.17%)
Weka r5178	267,707	2,068 (37.86%)	20,806 (35.95%)	1,195 5.74%	1,392 6.69%	5,219 25.08%	13,000 62.48%	315,092 (34.73%)	69,420 (35.83%)
Weka r10042	251,723	2,257 (28.93%)	19,773 (28.49%)	1,043 5.27%	1,254 6.34%	4,930 24.93%	12,546 63.45%	279,936 (25.13%)	64,760 (25.90%)
XML-Security	26,059	317 (73.82%)	2,353 (60.14%)	43 1.83%	96 4.08%	718 30.51%	1,496 63.58%	17,793 (56.02%)	6,273 (52.11%)

braces mean the coverage achieved by running the program test set. For Commons-Lang, 99.30% of classes and 93.03% methods were covered as well as 87.67% of duas and 91.64% of edges. The percentage of methods with a particular property (e.g., requiring more than 64 duas) with respect to the total number of methods is represented by the percentage number without braces. For example, Commons-Lang has 1.58% of its methods with more than 64 duas required. CF coverage were obtained with JaCoCo and DF coverage with BA-DUA. The lines of code were measure with CLOC.

Two programs — JTopas and Scimark2 — are classified as small systems. Three programs: Commons-Lang, Commons-Math 2.1 and XML-Security are medium size programs; and five programs — Commons-Math 3.2, HSQLDB, JFreeChart, Weka r5178 and r10042 — are large systems. With the exception of the programs for which we selected two versions, the others are the latest version as of September, 2013.

7.3 Treatments

The treatments were defined taking into account the coverage tool (if any) as follows:

Baseline. This treatment uses an uninstrumented version of the program.

JaCoCo. This treatment uses a JaCoCo instrumentated program tracking instructions and edges at run-time.

BA-DUA. This treatment uses a BA-DUA instrumentated program tracking duas at run-time.

7.4 Procedure

We collected two types of data: execution overhead and memory overhead. The execution overhead was determined by executing the test set accompanying each program. Excepting HSQLDB and JTopas, for which failing test cases were removed, the test set executed were those contained in the repository of the program. The memory overhead was obtained by the ratio of the total size of all instrumented classes by the size of all compiled classes of each program.

For each treatment, the execution time of the subject program was obtained by averaging the time of ten executions. It was collected using the `real` option of the Unix `time` command since we were interested in obtaining the wall clock time.

For two methods in Commons-Math 3.2 and two in HSLQDB the size of the instrumented method was larger than 64 kBytes which hampered their execution in the JVM. Our procedure was to did not instrument these methods. Thus, the dua coverage

of these methods was not obtained. JaCoCo and BA-DUA were run in off-line mode, that is, the instrumentation was performed before the execution of the programs. All data was collected using an Intel(R) Xeon(R) CPU W5580@3.20GHz, 2,042,732 kBytes of RAM, running on 64-Bit Linux Ubuntu Server with Java HotSpot(TM) 64-Bit Server VM (1.8.0_11).

7.5 Results

Table 16 – Execution overhead of the selected programs

Program	Baseline (s)	JaCoCo (s)	BA-DUA (s)
Commons-lang	12.40	12.12 -2.26%	12.15 -2.02%
Commons-Math 2.1	9.15	9.25 1.09%	11.12 21.53%
Commons-Math 3.2	81.72	89.88 9.99%	184.01 125.17%
HSQLDB	17.19	17.61 2.44%	18.64 8.44%
JFreeChart	2.92	2.89 -1.03%	2.90 -0.68%
JTopas	29.44	91.30 210.12%	35.88 21.88%
Scimark2	25.66	24.37 -5.03%	28.39 10.64%
Weka r5178	236.90	237.56 0.28%	397.73 67.89%
Weka r10042	49.20	46.35 -5.79%	59.72 21.38%
XML-Security	15.14	15.51 2.44%	16.67 10.11%

The execution overhead results are presented in Table 16. The table contains the absolute averaged execution times for each treatment. In addition, it comprises the overhead imposed by JaCoCo and BA-DUA in percentage terms. In the third row, the data regarding Commons-Math 2.1 is presented. The uninstrumented version of the program (Baseline) executed the test set in 9.15 seconds in average; the JaCoCo instrumented version executed in 9.25 seconds; and BA-DUA version in 11.12 seconds. The overhead imposed by JaCoCo was 1.09% and BA-DUA overhead was 21.53%.

In Figure 14, the ratios Baseline to JaCoCo and Baseline to BA-DUA are presented. The value 1 in the chart means that there is no overhead between the uninstrumented program and the one instrumented by the testing tool. On the other hand, a ratio equals to 1.5 means that there is 50% overhead when the instrumented version is executed. Figure 15

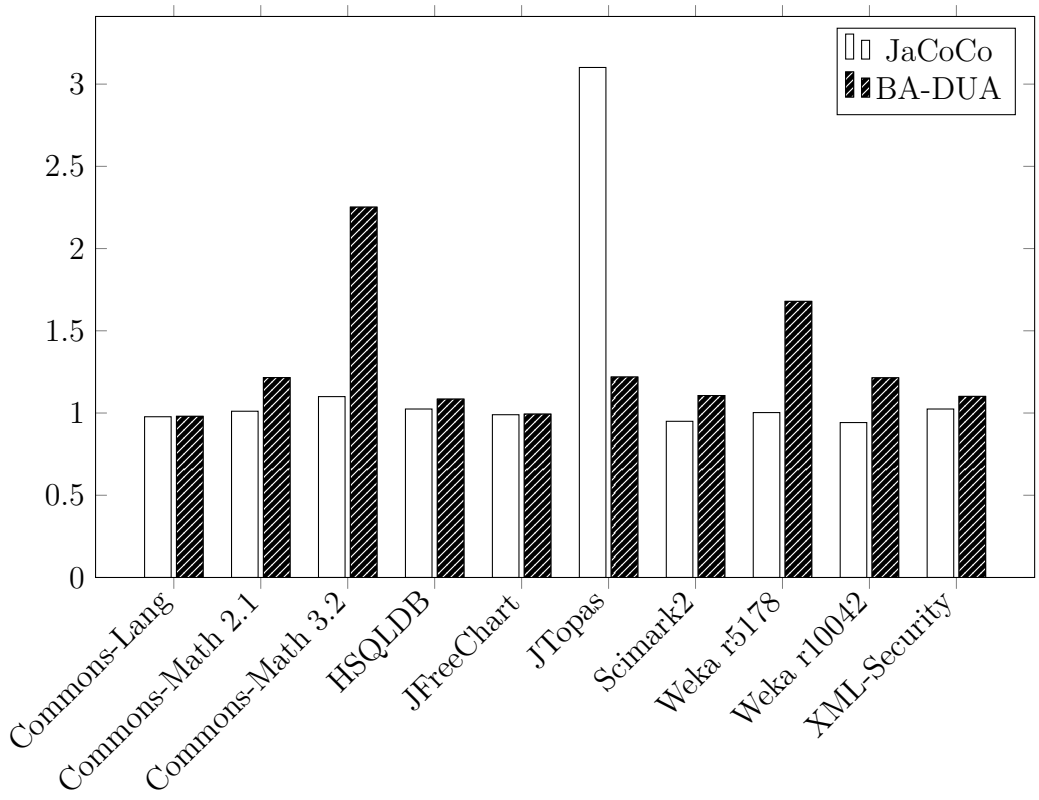


Figure 14 – JaCoCo and BA-DUA: Baseline execution overhead ratios

presents JaCoCo and BA-DUA memory overhead with respect to the Baseline; it describes the ratio between the size of all compiled classes of the uninstrumented program to the size of all JaCoCo instrumented classes and to all BA-DUA instrumented classes. Analogously, a value of 1 means that there is no memory overhead; a value of 2 means 100% of overhead.

7.6 Discussion

Our discussion of the results is guided by the research questions. The section is concluded with the threats to validity of the experiment.

7.6.1 BA penalties

The execution overhead imposed by BA varied from -2.02%, for Commons-Lang, to 125.17% for Commons-Math 3.2, with the average overhead being 28.43%. From Table 16 and Figure 14, for three of the subject programs, the overhead was less than 10%, which is negligible taking into account the number of duas tracked. The overhead was between 10 and 25% for five subjects; and only two subjects had an overhead over 25%. Thus, our data suggests that by using BA, DF testing can be applied to a larger class of programs with affordable execution overhead.

Misurda et al.'s demand-driven approach obtained an average of 127% overhead,

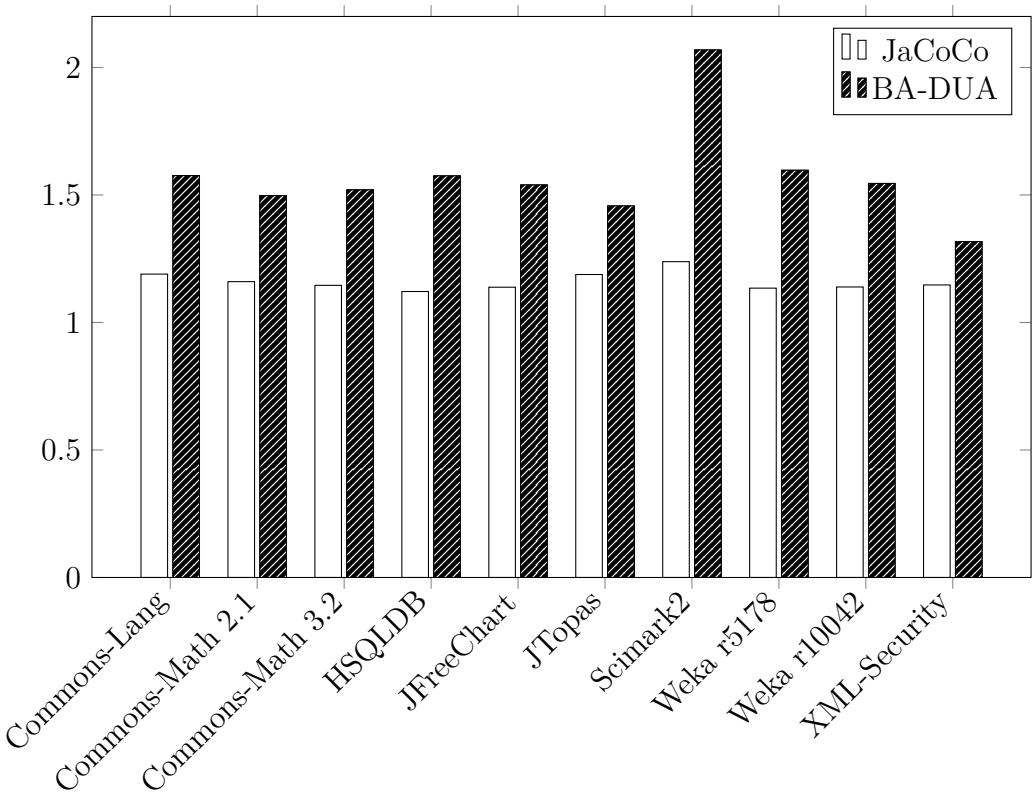


Figure 15 – JaCoCo BA-DUA: Baseline memory overhead ratios

the values varying from 4% to 279%. Hassan and Andrew report that DUA-FORENSICS imposed an overhead of 2078%, 160.4%, and 86.9% for JTopas, Scimark2, and XML-Security, respectively. Although a comparison with previous approaches should be carried out with caution due to differences in the DF testing implementation (e.g., inter- or intra-procedural testing) and JVM and hardware utilized, the results indicate that BA overhead is significantly smaller. This difference is more significant when comparing the size of the program utilized in experiments. Previously, only small to medium size programs were utilized. This is so because both approaches have to keep data structures that encompass all duas of the programs.

On other hand, BA performance is governed by the number of duas in each method and by how long the execution remains in particular methods. A BA-DUA bit vector is implemented as a 64-bit long; thus, there is no difference in tracking one or 64 duas. However, a method with 65 duas will require another long and more code to update the extra long. As a result, the execution overhead increases. Table 15 shows that the number of methods with more than 64 duas is small: at most 6.56% and as low as 1.58% of all methods. Even though, a small number of methods with high concentration of duas can hurt performance. For example, Commons-Math 3.2 has 285 methods with more that 64 duas; 25 of which with more than 320 duas (5 longs) and 5 with more than 1000 duas (16 longs). Thus, depending on how long the execution remains in these methods, the BA overhead will increase. Particularly, Commons-Math 3.2 has a high method coverage

(85.42%), which means that they are very likely to be executed by the test set.

BA memory overhead is essentially the bit vectors created (`covered`, `live`, and `cursleepy`) and the extra code inserted[4]. Figure 15 presents the program growth caused in BA-DUA instrumented programs in comparison to uninstrumented programs. With the exception of Scimark2, the memory overhead is around 50%. Although that can be considered a sizeable memory footprint, a BA-DUA instrumented program can run in the majority of hardware platforms, with the possible exception of embedded systems with very tight memory. However, the program growth can be reduced by instrumenting not nodes as in BA-DUA, but edges as in JaCoCo. This simple strategy will reduce the size of the instrumented program and, additionally, the execution overhead.

7.6.2 BA-DUA costs versus JaCoCo costs

A comparison with edge monitoring implemented in tools like JaCoCo is valid since dua coverage provided by BA-DUA can be approximately inferred from edge coverage. In terms of execution overhead, both tools have a similar performance for 4 out of 10 programs. The exceptions are Commons-Math, Scimark2 and Weka, for which JaCoCo is significantly better. For JTopas, BA-DUA performed better. BA-DUA instrumented Commons-Lang and JFreeChart outperform even the uninstrumented versions; this performance is possibly due to how the JVM Just-In-Time compiler optimizes register allocations and cache effectiveness for the BA-DUA instrumented code. JaCoCo outperform the uninstrumented version for 4 programs.

The JaCoCo JTopas instrumented version is hampered by the fact that a method without duas is called a huge amount of times. This method is not instrumented by BA-DUA; thus, it does not incur such an overhead. The decision of not instrumenting zero-duas methods is because dua coverage does not subsume edge coverage in the presence of infeasible paths [21]. Thus, one will not prescind a CF tool if method, edge and node coverage are needed. The results suggest, though, that for several programs the execution overhead is similar.

In applications in which DF testing is most indicated, namely, security and critical applications, approximate intra-procedural dua coverage will hardly be useful. The approximate dua coverage can overshoot the precise coverage in up to 30% [13]. In these scenarios, one is interested in knowing which paths were not tested; thus, a precise dua coverage is needed. It can be collected by using BA with an affordable overhead.

BA-DUA utilizes a bit to represent a dua whereas JaCoCo uses a boolean to represent an edge (node coverage is inferred from edge coverage). JaCoCo memory overhead varies from 12.14% (HSQLDB) to 23.84% (Scimark2), which can be observed by the almost

[4] This is an estimate since we are not taking into account the size of the global array of covered duas created at run-time for each loaded class.

constant ratio in Figure 15. BA-DUA incurs in higher memory overhead, but its growth should be analyzed by the number of requirements tracked at run-time. Scimark2, which is the worst case, with memory overhead of 106.86% for BA-DUA, requires around 5 times more duas to be tracked than edges. Thus, the memory overhead grows 5 times.

7.6.3 Threats to validity

The external validity of the experiment is arguable on the grounds that our large programs are not large enough. One may argue that a 50 KLOC program is not large or even a 250 KLOC program. Although systems with millions of lines of code are rather common, they are not monolithically coded. On the contrary, they are divided into manageable components of smaller size for which test sets are developed. Our goal was to identify programs that could be part of a system of more than 1 MLOC. In this sense, the medium size programs are also candidates to be a component of such systems. Our data suggests that BA-DUA can be applied to collect dua coverage of these components. However, it does not allow us to infer that BA-DUA can support integration testing of these huge systems.

The internal validity of our experiment concerns the results provided by JaCoCo and BA-DUA. JaCoCo is widely used in industry, which is an indication that its results are trustworthy. BA-DUA has been validated with small programs and its results were compared with those provided by JaBUTi. BA-DUA performs a more accurate DF analysis of bytecodes, but the results were similar.

Regarding conclusion and construction validities, the comparison among treatments was straightforward. In a first version of this work [47], a modest hardware was utilized to conduct the experiment. The average execution overhead was 38%. In this Master dissertation, we utilized a top hardware configuration and the execution overhead improved: in average, it was 28%. This shows that BA profits from more a efficient hardware; thus, the results are not biased when different hardware and JVM are utilized.

7.7 Final remarks

In this chapter, we presented an evaluation of the BA algorithm in the BA-DUA tool implementation. The next chapter contains the conclusions of the results achieved, our contributions, and the future work.

8 Conclusions

In this chapter, we present the summary of the results achieved, our contributions, and the future work.

8.1 Summary

The goal of this work was to show that Data-flow (DF) testing supporting tools can be implemented to tackle large programs. DF testing requires tests that traverse a path in which the definition of a variable and its subsequent use, i.e., a definition-use association (dua), is exercised.

To achieve such a goal, a tool called BA-DUA (Bitwise Algorithm-powered Definition-use Association coverage) was implemented. BA-DUA utilizes a strategy to track intra-procedural duas at run-time called Bitwise Algorithm (BA) [14].

BA was recently proposed and was only assessed by means of simulations. It is based on a simple idea which consists of encoding a solution for the reaching definitions data-flow problem [48] into an object code. In this research, we have described BA (Chapter 4) and how it is mapped into bytecodes (Chapters 5 and 6). BA-DUA implements such a mapping.

An experiment was conducted to assess the penalties imposed by BA (Chapter 7). Two kinds of penalties were assessed: execution overhead, given by the extra time needed to execute the instrumentation code; and memory overhead, represented by the growth of the object code due to the insertion of instrumentation code.

Ten programs with size varying from 800 LOC and 250 KLOC were utilized in the experiment. Two programs were classified as small (less than 5 KLOC), three as medium size programs (more than 5 KLOC and less than 50 KLOC), and five were categorized as large (more than 50 KLOC). The subject programs perform mathematical functions, data manipulation tasks, graphical functions and utilities functions for Java.

The BA-DUA execution overhead varied from negligible to 125%, with the average being 28%. However, only two subjects in ten programs had their performance impacted more than 25%; for three programs, the overhead was less than 10%. In terms of program growth, the BA memory overhead ranges from 32% to 107%, with the average being 57%.

The BA-DUA performance was also compared with that of the JaCoCo tool — a widely used control-flow (CF) testing supporting tool. In terms of execution overhead, both tools have a similar performance for 4 out of 10 programs. JaCoCo has a smaller

memory footprint, but tracks fewer requirements at run-time.

The results suggest that by using BA to track duas the overhead, especially the execution overhead, are significantly reduced. As a result, a larger class of programs can be assessed by intra-procedural DF testing. BA-DUA is available for download at <http://github.com/saeg/ba-dua/>.

8.2 Contributions

This research led to several contributions in the context of DF testing that are described below:

- A review of techniques and tools to support DF testing: We explored both the state of the art and the state of practice regarding techniques to track duas at run-time and DF testing tools in Chapter 3. Our bibliographical research indicates that the current techniques and tools do not scale for programs developed at industrial settings. This situation is due in part to the high cost of tracking duas at run-time.

- The Bitwise Algorithm: We devised BA to reduce the cost of tracking duas at run-time. Initially, it was evaluated by means of conservative simulations, which indicated that it was a promising strategy to track duas at run-time [14]. The implementation of BA in the BA-DUA tool showed that the algorithm performance is comparable to that of CF tools.

- ASM-DefUse and BA-DUA tools: Two tools were implemented to support the use of BA in programs developed at industrial settings. ASM-DefUse is an extension of the ASM library for analyzing and instrumenting programs compiled into bytecodes. It generates the duas required by the all-uses criterion for each method of a bytecode program. BA-DUA, in turn, utilizes ASM-DefUse to determine the duas and insert BA code in the program.

- DF testing scalability: The experiment presented in this work suggests that a DF testing tool implementing the BA strategy to track duas at run-time scales up well to medium and large programs.

8.3 Future work

Despite the promising results, there is room for improvement. The program growth and slowdown can be reduced by instrumenting edges, instead of nodes. However, the main drawback with BA are methods with a high concentration of duas since they require more code to be inserted, causing higher execution and memory overhead.

We intend to investigate how the subsumption relationship among duas [49] can be utilized to determine a minimal set of duas to be tracked at run-time. We conjecture that in methods with many duas this minimal set is relatively small. Other avenues of research encompass the extension of the BA to track inter-procedural duas and to tackle programs with multiple threads.

As a concluding remark, we hope that the results presented in this research encourage vendors to consider including DF testing in their set of testing tools.

Bibliography

1 MALDONADO, J. C. *Potential Uses Criteria: A Contribution to Structural Software Testing*. Thesis (Ph.D.) — State University of Campinas, 1991.

2 MYERS, G. J. *The Art of Software Testing*. New York: John Wiley & Sons, Inc., 1979.

3 RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: *Proceedings of the 6th International Conference on Software Engineering*. 1982.

4 RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 1985.

5 FEITELSON, D. G.; FRACHTENBERG, E.; BECK, K. L. Development and deployment at Facebook. *IEEE Internet Computing*, 2013.

6 GRIMM, S. Facebook engineering. What kind of automated testing does Facebook do? Available in: <http://www.quora.com/Facebook-Engineering/What-kind-of-automated-testing-does-Facebook-do>.

7 MOIR, K. SDK code coverage with JaCoCo. Blog post. Available in: <http://relengofthenerds.blogspot.com.br/2011/03/sdk-code-coverage-with-jacoco.html>.

8 HUTCHINS, M.; FOSTER, H.; GORADIA, T.; OSTRAND, T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: *Proceedings of the 16th International Conference on Software Engineering*. 1994.

9 FRANKL, P. G.; IAKOUNENKO, O. Further empirical studies of test effectiveness. In: *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 1998.

10 DAO, T.-B.; SHIBAYAMA, E. Security sensitive data flow coverage criterion for automatic security testing of web applications. In: *Proceedings of the 3rd International Conference on Engineering Secure Software and Systems*. 2011.

11 SANTELICES, R.; JONES, J. A.; YU, Y.; HARROLD, M. J. Lightweight fault-localization using multiple coverage types. In: *Proceedings of the 31st International Conference on Software Engineering*. 2009.

12 MISURDA, J.; CLAUSE, J. A.; REED, J. L.; CHILDERS, B. R.; SOFFA, M. L. Demand-driven structural testing with dynamic instrumentation. In: *Proceedings of the 27th International Conference on Software Engineering*. 2005.

13 SANTELICES, R.; HARROLD, M. J. Efficiently monitoring data-flow test coverage. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. 2007.

14 CHAIM, M. L.; ARAUJO, R. P. A. An efficient bitwise algorithm for intra-procedural data-flow testing coverage. *Information Processing Letters*, 2013.

15 DELAMARO, M. E.; CHAIM, M. L.; VINCENZI, A. M. R. Técnicas e ferramentas de teste de software. In: *Atualizações em informática 2010 (JAI 2010)*. Rio de Janeiro: Editora PUC-Rio, 2010.

16 BALL, T.; LARUS, J. R. Efficient path profiling. In: *Proceedings of the 29th Annual International Symposium on Microarchitecture*. 1996.

17 AGRAWAL, H. Efficient coverage testing using global dominator graphs. *ACM SIGSOFT Software Engineering Notes*, 1999.

18 TIKIR, M. M.; HOLLINGSWORTH, J. K. Efficient online computation of statement coverage. *Journal of Systems and Software*, 2005.

19 SOFFA, M. L.; WALCOTT, K. R.; MARS, J. Exploiting hardware advances for software testing and debugging. In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011.

20 FISCHMEISTER, S.; BA, Y. Sampling-based program execution monitoring. *ACM SIGPLAN Notices*, 2010.

21 FRANKL, P. G.; WEYUKER, E. J. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 1988.

22 MISURDA, J.; CLAUSE, J.; REED, J.; CHILDERS, B. R.; SOFFA, M. L. Jazz: a tool for demand-driven structural testing. In: *Proceedings of the 14th International Conference on Compiler Construction*. 2005.

23 JaCoCo web site. <http://www.eclemma.org/jacoco/>. Accessed: 06/07/2014.

24 Clover web site. <http://www.atlassian.com/software/clover/>. Accessed: 06/07/2014.

25 Cobertura web site. <http://cobertura.sourceforge.net/>. Accessed: 06/07/2014.

26 EMMA web site. <http://emma.sourceforge.net/>. Accessed: 06/07/2014.

27 CodeCover web site. <http://codecover.org/>. Accessed: 06/07/2014.

28 Quilt web site. <http://quilt.sourceforge.net/>. Accessed: 06/07/2014.

29 JVMDI web site. <http://jvmdicover.sourceforge.net/>. Accessed: 06/07/2014.

30 GroboCoverage web site. <http://groboutils.sourceforge.net/codecoverage/>. Accessed: 06/07/2014.

31 OSTRAND, T. J.; WEYUKER, E. J. Data flow-based test adequacy analysis for languages with pointers. In: *Proceedings of the Symposium on Testing, Analysis, and Verification*. 1991.

32 HORGAN, J. R.; LONDON, S. A data flow coverage testing tool for C. In: *Proceedings of 2nd Symposium on Assessment of Quality Software Development Tools*. 1992.

33 CHAIM, M. L. *POKE-TOOL — A Tool to support Structural Program Testing based on Data Flow Analysis*. Dissertation (M.Sc) — State University of Campinas, 1991.

34 VINCENZI, A. M. R.; MALDONADO, J. C.; WONG, W. E.; DELAMARO, M. E. Coverage testing of Java programs and components. *Science of Computer Programming*, 2005.

35 BLUEMKE, I.; REMBISZEWSKI, A. Dataflow approach to testing Java programs. In: *Proceedings of the 4th International Conference on Dependability of Computer Systems*. 2009.

36 SEESING, A.; ORSO, A. InsectJ: a generic instrumentation framework for collecting dynamic information within Eclipse. In: *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*. 2005.

37 Coverlipse web site. <http://coverlipse.sourceforge.net/>. Accessed: 06/07/2014.

38 JMockit web site. <http://jmockit.github.io/>. Accessed: 06/07/2014.

39 MALDONADO, J. C.; CHAIM, M. L.; JINO, M. Test case selection based on the potential uses criteria. In: *Proceedings of the II Brazilian Symposium on Software Engineering*. 1988.

40 VALLÉE-RAI, R.; HENDREN, L.; SUNDARESAN, V.; LAM, P.; GAGNON, E.; CO, P. Soot - a Java optimization framework. In: *Proceedings of CASCON 1999*. 1999.

41 YANG, Q.; LI, J. J.; WEISS, D. M. A survey of coverage-based testing tools. *The Computer Journal*, 2009.

42 HASSAN, M. M.; ANDREWS, J. H. Comparing multi-point stride coverage and dataflow coverage. In: *Proc. of 35th International Conference on Software Engineering*. 2013.

43 YAN, J.; ZHANG, J. An efficient method to generate feasible paths for basis path testing. *Information Processing Letters*, 2008.

44 ARAUJO, R. P. A.; ACCIOLY, A.; ALENCAR, F. A.; CHAIM, M. L. Evaluating instrumentation strategies by program simulation. In: *Proceedings of IADIS International Conference on Applied Computing*. 2011.

45 LINDHOLM, T.; YELLIN, F.; BRACHA, G.; BUCKLEY, A. *The Java (R) Virtual Machine Specification, (Java SE 8 Edition)*. 2014.

46 BRUNETON, E. *ASM 4.0 A Java bytecode engineering library*. 2011.

47 ARAUJO, R. P. A. de; CHAIM, M. L. Data-flow testing in the large. In: *7th International Conference on Software Testing, Verification and Validation*. 2014.

48 AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. *Compilers: principles, techniques, and tools*. Boston: Pearson Addison-Wesley, 2007.

49 MARRÉ, M.; BERTOLINO, A. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, 2003.

Appendix

APPENDIX A – Correctness proof of Algorithm 1

Theorem 1 *Algorithm 1 is correct, that is, the* ***Covered*** *set contains the duas exercised in path $s \ldots l$ where s is the start node and l is the last node traversed in a program execution.*

Proof. We prove the correctness of Algorithm 1 by induction.

Basis step. The base case consists of determining the values of **Covered**, **Alive**, and **CurSleepy** for a path with a single node traversed and equal to s, the start node. For such a path, **Covered** is empty because duas (d, s, X) or (d, (u',s), X) do not exist. **Alive** contains duas (s, u, X) and (s, (u',u), X) which are born in s and **CurSleepy** comprises all p-use duas, except the p-use duas (s, (s,u), X).

Basis proof. After line 6, **Covered** is empty because the result of **Alive** minus **CurSleepy** is intersected with **PotCovered(s)** whose value is empty because there is no dua with a use in s. **Alive** is set up with duas (s, u, X) and (s, (u',u), X) after line 7 since its initial empty value is added to **Born(s)**, which contains duas (s, u, X) and (s, (u',u), X). **CurSleepy** will receive the **Sleepy(s)**, determined according to the definition of **Sleepy(n)** in Chapter 4, after line 8. Thus, the basis step is proved.

Induction. Assuming that **Covered** contains the exercised duas and **Alive** the *alive* duas in path ($s \ldots n_k$) and **CurSleepy** contains the value of **Sleepy(n_k)**, then, after node n_{k+1} is processed in Algorithm 1, **Covered** will contain the exercised duas and **Alive** the alive duas in path ($s \ldots n_k, n_{k+1}$) and **CurSleepy** will contain the value of **Sleepy(n_{k+1})**.

Induction proof. Firstly, we show that **Covered** is correct after n_{k+1} is processed regarding only c-use duas. Following the inductive step, **Covered** and **Alive** are correct in path ($s \ldots n_k$). At line 6 of Algorithm 1, the new **Covered** will contain the previous **Covered** added with the intersection of **Alive** and **PotCovered(n_{k+1})**, which contains c-use duas (d, n_{k+1}, X), according to the definition in Chapter 4. **CurSleepy** does not influence the result of **Covered** regarding c-use duas since it contains just p-use duas. Hence, **Covered** will contain the covered c-use duas up to n_k plus c-use duas (d, n_{k+1}, X) that are present both in **Alive** and **PotCovered(n_{k+1})**; that is, those c-use duas that are covered when path ($s \ldots n_k$, n_{k+1}) is traversed.

The rationale with respect to p-use duas is similar. At line 6, **CurSleepy** is subtracted from **Alive**. However, **CurSleepy** is equal to **Sleepy(n_k)** before n_{k+1} is processed due to the inductive step. This implies **CurSleepy** contains all duas (d, (u',u),

X) such that $u' \neq n_k$, following the definition of **Sleepy**(n) in Chapter 4. In other words, only p-use duas (d, (n_k,u), X) *are not* in **CurSleepy** before n_{k+1} is processed. Thus, (d, (n_k,u), X) are the only p-use duas left after the subtraction of **CurSleepy** from **Alive**. P-use duas (d, (n_k,u), X) are then intersected with **PotCovered**(n_{k+1}), which contains p-use duas (d, (u',n_{k+1}), X) (see definition in Chapter 4). As a result, **Covered** will contain the covered p-use duas up to n_k plus p-use duas (d, (n_k,n_{k+1}), X) that are present both in **Alive** and in **PotCovered**(n_{k+1}); that is, those p-use duas that are covered when path (s ... n_k, n_{k+1}) is traversed.

To complete the induction proof, however, it needs to be shown that **Alive** and **CurSleepy** are correct after n_{k+1} is processed. Considering the **Alive** working set, its value is correct up to n_k before executing line 7 of the algorithm. At line 7, duas (d, u, X) or (d, (u',u), X), such that X is defined in n_{k+1} and $d \neq n_{k+1}$, are eliminated from **Alive** because they belong to **Disabled**(n_{k+1}) — see definition of **Disabled**(n) in Chapter 4. This subtraction operation represents the redefinition of variable X at n_{k+1}. The subsequent union with **Born**(n_{k+1}) which contains duas (n_{k+1}, u, X) and (n_{k+1}, (u',u), X), adds to **Alive** those duas created from the definition of X at n_{k+1}. Thus, after line 7, **Alive** contains the duas *alive* in path (s ... n_k, n_{k+1}). At line 8, **CurSleepy** receives the value of **Sleepy**(n_{k+1}). Hence, the induction is proved.

After $n_{k+1} = l$ is processed, **Covered** will contain the set of duas exercised in path ($s \ldots l$), which proves Theorem 1. ■

Annex

ANNEX A – JVM instruction set

Constants	**Loads**	**Stores**
nop	iload	istore
aconst_null	lload	lstore
iconst_m1	fload	fstore
iconst_0	dload	dstore
iconst_1	aload	astore
iconst_2	iload_0	istore_0
iconst_3	iload_1	istore_1
iconst_4	iload_2	istore_2
iconst_5	iload_3	istore_3
lconst_0	lload_0	lstore_0
lconst_1	lload_1	lstore_1
fconst_0	lload_2	lstore_2
fconst_1	lload_3	lstore_3
fconst_2	fload_0	fstore_0
dconst_0	fload_1	fstore_1
dconst_1	fload_2	fstore_2
bipush	fload_3	fstore_3
sipush	dload_0	dstore_0
ldc	dload_1	dstore_1
ldc_w	dload_2	dstore_2
ldc2_w	dload_3	dstore_3
	aload_0	astore_0
	aload_1	astore_1
	aload_2	astore_2
	aload_3	astore_3
	iaload	iastore
	laload	lastore
	faload	fastore
	daload	dastore
	aaload	aastore
	baload	bastore
	caload	castore
	saload	sastore

Stack	**Math**	**Conversions**
pop	iadd	i2l
pop2	ladd	i2f
dup	fadd	i2d
dup_x1	dadd	l2i
dup_x2	isub	l2f
dup2	lsub	l2d
dup2_x1	fsub	f2i
dup2_x2	dsub	f2l
swap	imul	f2d
	lmul	d2i
	fmul	d2l
	dmul	d2f
	idiv	i2b
	ldiv	i2c
	fdiv	i2s
	ddiv	
	irem	
	lrem	
	frem	
	drem	
	ineg	
	lneg	
	fneg	
	dneg	
	ishl	
	lshl	
	ishr	
	lshr	
	iushr	
	lushr	
	iand	
	land	
	ior	
	lor	
	ixor	
	lxor	
	iinc	

Comparisons
lcmp
fcmpl
fcmpg
dcmpl
dcmpg
ifeq
ifne
iflt
ifge
ifgt
ifle
if_icmpeq
if_icmpne
if_icmplt
if_icmpge
if_icmpgt
if_icmple
if_acmpeq
if_acmpne

Control
goto
jsr
ret
tableswitch
lookupswitch
ireturn
lreturn
freturn
dreturn
areturn
return

References
getstatic
putstatic
getfield
putfield
invokevirtual
invokespecial
invokestatic
invokeinterface
invokedynamic
new
newarray
anewarray
arraylength
athrow
checkcast
instanceof
monitorenter
monitorexit

Extended
wide
multianewarray
ifnull
ifnonnull
goto_w
jsr_w

Reserved
breakpoint
impdep1
impdep2