Using Dynamic Analysis Of Java Bytecode For Evolutionary Object-Oriented Unit Testing

José Carlos Bregieiro Ribeiro[1], Francisco Fernández de Vega[2], Mário Zenha-Rela[3]

[1]Polytechnic Institute of Leiria (IPL)
Campus 2, Morro do Lena, Alto do Vieiro – Leiria – Portugal

[2]University of Extremadura (UNEX)
C/ Sta Teresa de Jornet, 38 – Mérida – Spain

[3]University of Coimbra (UC)
CISUC, Department of Informatics Engineering, P 3030-290 – Coimbra – Portugal
jose.ribeiro@estg.ipleiria.pt, fcofdez@unex.es, mzrela@dei.uc.pt

Abstract. *The focus of this paper is on presenting a methodology for generating and optimizing test data by employing evolutionary search techniques, with basis on the information provided by the analysis and interpretation of Java bytecode and on the dynamic execution of the instrumented test object.*

The main reason to work at the bytecode level is that even when the source code is unavailable, structural testing requirements can still be derived and used to assess the quality of a given test set and to guide the evolutionary search towards reaching specific test goals.

Java bytecode retains enough high-level information about the original source code for an underlying model for program representation to be built. The observations required to select or generate test data are obtained by employing dynamic analysis techniques – i.e. by instrumenting, tracing and analysing Java bytecode.

1. Introduction

Software testing is an expensive process, typically consuming roughly half of the total costs involved in the software development process while adding nothing to the raw functionality of the final product. Yet, it remains the primary method through which confidence in software is achieved. In industry, this process is often done manually – with the responsibility of assessing the quality of a given software product usually falling on the software tester. However, locating suitable test data can be time-consuming, difficult and expensive; automation of test data generation is, therefore, vital to advance the state-of-the-art in software testing.

Test data selection, generation and optimization deals with locating good test data for a particular test criterion. The application of evolutionary algorithms to test data generation is often referred to in literature as Evolutionary Testing (Mantere and Alander 2005). In evolutionary testing, meta-heuristic search techniques are employed to select or generate test data. The search space is the input domain of the test object,

and the problem is to find a (minimal) set of input data – test cases – that satisfies a certain test criterion. In particular case of object-oriented programs, a sequence of method invocations is required to cover the test goal, and the sequence search space is an explosive space. The application of search-based strategies for object-oriented unit testing has not yet been investigated comprehensively.

In this paper, we present an approach for guiding the evolutionary search towards generating test sets using coverage metrics derived from the test object's Java bytecode. The main reason to work at the bytecode level is that even when the test object's source code is unavailable, structural testing requirements can still be derived and used to assess the quality of a given test set. The observations required to extract such metric are obtained by employing dynamic analysis techniques – i.e. by instrumenting, tracing and analysing Java bytecode.

In the following section, background on the topics of testing methodologies, quality criteria, evolutionary search techniques and fitness evaluation is provided; related work is reviewed in Section 4. In section 5, we present our methodology for employing dynamic analysis of Java bytecode for test quality assessment and optimization and, on Section 6 the complete framework of our tool is outlined. The concluding chapter resumes the key ideas of this paper and presents some topics for future research.

2. Background

The assessment of the quality of a given test set can be achieved functionally (black-box testing) or structurally (white-box testing). Black-box testing is concerned with showing the conformity between the implementation and its functional specification; with white-box testing techniques, test case design is performed with basis on the program structure. Black-box testing is the most widely used testing approach; however, its applicability is often hindered by the need for a formal specification of the test object to be available. With white-box testing, the metrics for measuring the thoroughness of a given test set can be extracted from the structure of the target's source code, or even from compiled code (e.g. Java bytecode).

Traditional white-box criteria include structural (e.g. statement, branch) coverage and data flow coverage. The basic idea is to ensure that all of the control elements in a program are executed by a given test set, providing evidence of the quality of the testing activity; a test set that contains test cases that exercise all such elements is said to be adequate with respect to the corresponding criterion.

The evaluation of the quality of a given test set and the guidance to the test case selection using white-box criteria generally requires the definition of an underlying model for program representation – usually a control-flow graph (CFG). The CFG is an abstract representation of a given method in a class; control-flow testing criteria can be derived based on such a program representation to provide a theoretical and systematic mechanism to select and assess the quality of a given test set.

Two well known control-flow testing standards to derive testing requirements from the CFG are the all-nodes and all-edges criteria (Vincenzi, Delamaro et al. 2006). The all-nodes criterion requires that each node of a given CFG is executed at least once. To distinguish between instructions that are executed under the normal execution of the

program from others that require an exception to be executed, this criterion can be subdivided into two non-overlapping testing criteria so that the testing activity can focus on different aspects of a program at a time:

- all-nodes-exception-independent (All-Nodes$_{ei}$): requires every node of the CFG reachable through an exception-free path to be executed at least once.
- all-nodes-exception-dependent (All-Nodes$_{ed}$): requires every node of the CFG not reachable through an exception-free path to be executed at least once.

Conversely, the all-edges criterion requires that each control-flow deviation is executed at least once. To consider the control-flow in relation to the exception-handling mechanism, this criterion also is subdivided into two non-overlapping testing criteria: all-edges-exception-independent (All-Edges$_{ei}$) and all-edges-exception-dependent (All-Edges$_{ed}$).

The observations needed to assemble the metrics required by these criteria can be collected by abstracting and modelling the behaviours programs exhibit during execution – either by static or dynamic analysis techniques (Tracey, Clark et al. 2002). Dynamic analysis involves executing the actual test object and monitoring its behaviour; while it may not possible to draw general conclusions from dynamic analysis, it provides evidence of the successful operation of the software. In contrast, static analysis involves the construction and analysis of an abstract mathematical model of the system (e.g. symbolic execution). Static analysis is performed without executing the method under test, but rather this abstract model; this type of analysis is complex, and often incomplete due to the simplifications in the model.

If dynamic analysis techniques are employed, the ability to observe program execution is paramount. Events that need to be captured range from simple observations – such as execution of structural entities – to more complex examinations – such as thread and object creation, field manipulations, and object locking behaviour (Kinneer, Dwyer et al. 2006). Dynamic monitoring for events in Java software can be achieved through instrumentation of Java bytecode.

Bytecode is an assembly-like language that retains much of the high-level information about the original source program. Class files (i.e. compiled Java programs containing bytecode information) are a portable binary representation that contains class related data such as the class name, its superclass name, information about the variables and constants, and the bytecode instructions of each method (Vincenzi, Maldonado et al. 2005). Using bytecode as the basis for building the CFG allows broadening the scope of applicability of software testing tools, since the target object's source code is often unavailable; it can be used, for instance, to perform structural testing on third party Java components. In addition, the bytecode can be seen as an intermediate language, so the analysis performed at this level can be mapped back to the original high-level language that generated the bytecode.

Evolutionary algorithms have been used successfully for the unit testing of procedural software, and their application to the generation of quality test data for object-oriented software is an active field of research. Within the paradigm of object-orientation, the major concept is the object – which possesses attributes, constructors and methods. A test case for object-oriented software does not comprise only numerical

test data; a sequence of constructor and method calls is also necessary. Usually, multiple objects are involved in one single test case (Wappler and Lammermann 2005):

- At the least, an instance of the class under test is needed.
- Additional objects, which are required (as parameters) for the creation of the object under test and for the invocation of the method under test, must be available. Again, for the creation of these additional objects, more additional objects may be required.
- Depending on the kind of test, the participating objects may have to be put into particular states in order for the test scenario to be processed in the desired way. Consequently, method calls must be issued for these objects.

A fitness function for object-oriented evolutionary testing must evaluate test cases according to their ability to meet a given test goal. If white-box criteria are employed, the CFG and the monitored execution of the test object are used to access the adequateness of test cases – i.e. if the CFG node and/or path defined as the test goal was exercised by the execution of a particular test case over the test object.

In (Wappler and Wegener 2006a) a distance-based fitness function, which expresses how close the execution of a test case over the test object is to reaching the current test goal, was proposed. This closeness is expressed in terms of three distances:

- The Method Call Distance (`dMC`): expresses how close the test case execution approached the method under test in terms of the number of methods called. In case of a runtime exception, execution of a method call sequence terminates prematurely, meaning that the method under test is not called.
- The Control Node Distance (`dCN`): expresses how close execution of the test object approached the target CFG node.
- The Local Problem Node Distance (`dPN`): expresses how far the test object's execution is away from diverging along the branch of the problem node which leads to the test goal.

The metric `dMC` works at the test case level, and steers the evolutionary search towards producing feasible test cases – i.e. it ensures that a method call sequence of a given test case generates no runtime exceptions that prevent the method under test from being called.

Metrics `dCN` and `dPN`, on the other hand, are employed to cover individual test goals on the test object, and are computed with basis on the CFG. In (Wegener, Baresel et al. 2001), four methodologies – which depend on the CFG and the required test purpose – for guiding the evolutionary search toward reaching particular test goals were outlined, and the corresponding fitness functions were described:

- Node-oriented methods: require the attainment of specific nodes in the CFG (e.g. statement test, condition test).
- Path-oriented methods: require the execution of certain paths in the CFG (e.g. path tests).

- Node-path-oriented methods: require the achievement of a specific node and, from this node, the achievement of a specific path through the CFG (e.g. branch test, segment coverage).
- Node-node-oriented methods: aim to execute program paths that cover certain node combinations of the CFG in a pre-determined sequence without specifying a specific path between nodes (e.g. data-flow criteria).

3. Related Work

Interesting review articles on the topic of Evolutionary Testing include that of McMinn (McMinn 2004), who presents a review of meta-heuristic techniques that have been used in software test data generation, namely Hill Climbing, Simulated Annealing and – most interestingly – Evolutionary Algorithms. Namely, the main achievements in automating test data generation in the areas of structural testing, functional testing, and grey-box testing are summarized. In (Mantere and Alander 2005) an in-depth index of the work developed in the area is provided; topics include genetic algorithms applied to coverage testing, test data generation, testing program dynamics, black-box testing and software quality.

Both works pinpoint the state problem (McMinn and Holcombe 2003) as the main issue to be faced by researchers in this field. It occurs with methods that exhibit state-like qualities by storing information in internal variables; such variables are hidden from the optimization process because they are not available to external manipulation. The only way to change the values of these variables is through execution of statements that perform assignments to them. In object-oriented software this can occur through the use of variables that are protected from external manipulation using access modifiers.

The first approach to the field of evolutionary testing for object-oriented software was presented in (Tonella 2004); in this work, input sequences were generated using evolutionary algorithms for the white-box testing of classes. Genetic algorithms were the evolutionary approach employed, with potential solutions (test cases) being represented as chromosomes. A source-code representation was used, and an original evolutionary algorithm – with special evolutionary operators for recombination and mutation on a statement level (i.e. mutation operators insert or remove methods from a test program) – was defined. A population of individuals, representing the test cases, was evolved in order to increase a measure of fitness, accounting for the ability of the test cases to satisfy a coverage criterion of choice – the proportion of all control and call dependences that lead to the given target. New test cases are generated as long as there are targets to be covered or a maximum execution time is reached. However, the encapsulation problem was not addressed, and this proposal only dealt with a simple state problem; additionally, with this approach, Universal Evolutionary Algorithms – evolutionary algorithms, provided by popular toolboxes, which are independent from the application domain and offer a variety of predefined, probabilistically well-proven evolutionary operators – cannot be applied.

An approach which built upon an Ant Colony Optimization Algorithm was presented by (Liu, Wang et al. 2005). The focus was on the generation of the shortest method call sequence for a given test goal, under the constraint of state dependent

behaviour and without violating encapsulation. Ant PathFinder, hybridizing Ant Colony Optimization and Multiagent Genetic Algorithms were employed. To cover those branches enclosed in private/protected methods without violating encapsulation, call chain analysis on class call graphs was introduced.

In (Wappler and Lammermann 2005) an approach for the automatic generation of test programs for object-oriented unit testing was presented, focusing on the usage of Universal Evolutionary Algorithms. An encoding was proposed that represented object-oriented test programs as basic type value structures, allowing for the application of various search-based optimization techniques such as Hill Climbing or Simulated Annealing. The generated test programs could be transformed into test classes according to popular testing frameworks. The suggested encoding, however, did not prevent the generation of individuals which could not be decoded into test programs without errors; their fitness function used different penalty mechanisms in order to penalize invalid sequences and to guide the search towards regions that contained valid sequences. Due to the generation of infeasible sequences, the approach lacked efficiency for more complicated cases.

In (Wappler and Wegener 2006b) a different approach to the subject was presented. Potential solutions were encoded using a Strongly-Typed Genetic Programming (STGP) methodology, with method call sequences being represented by method call trees; these trees are able to express the call dependences of the methods that are relevant for a given test object. To account for polymorphic relationships which exist due to inheritance relations, the STGP types used by the function set are specified in correspondence to the type hierarchy of the test cluster classes. The emphasis of this work is on sequence feasibility; the usage of STGP preserves feasibility throughout the entire search process. The fitness function does need, however, to incorporate a penalty mechanism for test cases which include method call sequences that generate runtime exceptions. The issue of runtime exceptions was precisely the main topic in (Wappler and Wegener 2006a). This methodology yielded very encouraging results. For a simple custom-tailored test cluster, the set of generated test cases achieved full (100%) branch coverage: during the search, 11966 test programs were generated and evaluated, and the resulting test set contained 3 test cases; a control run, in which random test cases where produced for comparison purposes, stopped after having evaluated 43233 test programs (in accordance to the specified termination criteria), and the generated test set achieved a coverage of 66%. In a more complex scenario, four classes where tested and full coverage was achieved for all of the test objects.

In the abovementioned approaches, the underlying model for program representation is built with basis on the test object's source-code; moreover, instrumentation of the test object for extracting tracing information is also done at the source-code level. To the best of our knowledge, there are no evolutionary approaches to the unit-testing of object-oriented software that employ dynamic bytecode analysis to derive structural testing criteria.

The application of evolutionary algorithms and bytecode analysis for test automation was, however, already studied in different scenarios. In (Cheon, Kim et al. 2005) an attempt to automate unit testing of object-oriented programs is described. A black-box approach for investigating the use of genetic algorithms for test data generation is employed, and program specifications written in JML are used for test

result determination. The JML compiler was extended to make Java bytecode produce test coverage information. In (Muller, Lembeck et al. 2004), the layout of a symbolic Java virtual machine (SJVM), which discovers test cases using a definable structural coverage criterion with basis on static analysis techniques, is described. Java bytecode is executed symbolically, and the decision whether to enter a branch or throw an exception is based on the earlier constraints, a constraint solver and current testing criterion. The SJVM has been implemented in a test tool called GlassTT. This work, however, doesn't address exception-related and method interaction-related criteria, and only procedural software scenarios are described.

4. Dynamic Analysis Of Java Bytecode For Test Quality Optimization

The focus of this paper is on presenting a methodology for generating and optimizing test data by employing evolutionary search techniques, solely with basis on the information provided by the analysis Java bytecode and on the dynamic execution of the instrumented test object's class files.

In this section, the simple test cluster defined in (Wappler and Wegener 2006a) is used for demonstration purposes. We focus on the `Controller.reconfigure(Config)` public method; its source code is reproduced in Figure 1.

```
public void reconfigure(Config cfg) throws Exception {
    if( cfg.getSignalCount() > MAX_SIGNALS )
        throw new Exception("Too many signals.");
    if(cfg.getPort()<MIN_PORT||cfg.getPort()>MAX_PORT)
        throw new Exception("Invalid port.");
    this.cfg = cfg;
    signals = new int[cfg.getSignalCount()];
}
```

Figure 1. Source code for the method `Controller.reconfigure(Config)`

The source code provided by this example was compiled using JDK 1.5. The bytecode instructions of the compiled `Controller.reconfigure(Config)` public method are depicted in Figure 2.

4.1. Bytecode Analysis

In order to understand the details of Java bytecode, a preliminary discussion on how the Java virtual machine (Lindholm and Yellin 1999) works regarding the execution of the bytecode must take place. A JVM is a stack-based machine; each thread has a JVM stack which stores *frames*. A frame is created each time a method is invoked, and consists of an operand stack, an array of local variables, and a reference to the runtime constant pool of the class of the current method (Haggar 2001).

The array *of local variables* contains the parameters of the method and the values of the local variables. The size of the array of local variables is determined at compile time, and is dependent on the number and size of local variables and formal method parameters. The parameters are stored first, beginning at index 0. If the frame is for a constructor or an instance method, the `this` reference is stored at location 0; location 1 contains the first formal parameter, location 2 the second, and so on. For a static method, the first formal method parameter is stored in location 0, the second in location 1, and so on. The *operand stack* is a LIFO stack used to push and pop values;

its size is also determined at compile time. Certain bytecode instructions push values onto the operand stack; others take operands from the stack, manipulate them, and push the result. The operand stack is also used to receive return values from methods.

In Figure 2, The `aload_1` instruction at location 0 pushes the value from the index 1 of local variable table onto the operand stack – i.e. it pushes the parameter `cfg` of the method `Controller.reconfigure(Config cfg)` onto the top of the operand stack (a reference to an object of type `Config`). The `invokevirtual` instruction at location 1 invokes the instance method `Config.getSignalCount()` on the object `cfg` (popped from the top of the operand stack); the value returned by this method is pushed onto the top of the operand stack. The `iconst_5` instruction at location 4 loads the integer value `5` onto the top of the operand stack. At this point, the operand stack contains two values: the integer `5` on top, and the value returned by the `Config.getSignalCount()` on the bottom. The `if_icmple` instruction loads both those values from the operand stack, and compares them: if `5` is lower than or equal to the value returned from the `getSignalCount` method, instruction flow is transferred to instruction 18.

```
cfg.Controller.public_void_reconfigure(cfg.Config_cfg)
_throws_java.lang.Exception
Code(max_stack = 3, max_locals = 2, code_length = 64)
0:    aload_1
1:    invokevirtual cfg.Config.getSignalCount ()I (6)
4:    iconst_5
5:    if_icmple #18
8:    new <java.lang.Exception> (7)
11:   dup
12:   ldc "Too many signals." (8)
14:   invokespecial java.lang.Exception (java.lang.String)
17:   athrow
18:   aload_1
19:   invokevirtual cfg.Config.getPort ()I (10)
22:   sipush 8000
25:   if_icmplt #38
28:   aload_1
29:   invokevirtual cfg.Config.getPort ()I (10)
32:   sipush 8005
35:   if_icmple #48
38:   new <java.lang.Exception> (7)
41:   dup
42:   ldc "Invalid port." (11)
44:   invokespecial java.lang.Exception (java.lang.String)
47:   athrow
48:   aload_0
49:   aload_1
50:   putfield cfg.Controller.cfg Lcfg/Config; (2)
53:   aload_0
54:   aload_1
55:   invokevirtual cfg.Config.getSignalCount ()I (6)
58:   newarray <int>
60:   putfield cfg.Controller.signals [I (3)
63:   return
```

Figure 2. Java bytecode for the method `Controller.reconfigure(Config)`

This brief analysis helps to support the following conclusions: firstly, the bytecode instructions contain enough high-level information for coverage criteria to be applied at the bytecode level; secondly, it is possible to group some instructions into a smaller set of basic blocks that can ease the representation of the compiled program using a CFG and, consequently, the application of dynamic analysis and structural coverage metrics on the target object.

The purpose of the `aload_1` instruction is, in fact, to prepare the operand stack for the `getSignalCount` method call at location 2. Equally, the `iconst_5` instruction prepares the stack for the `if` instruction on location 5. We group these instructions into two basic blocks: the first pair is grouped into a Call block; the second pair is grouped into a Basic Instruction block of the sub-type “if”.

4.2. CFG Definition and Interpretation

In our approach, bytecode instructions are grouped into a set of basic blocks – namely, Basic Instruction blocks and Call Blocks. These blocks cover the core of nodes required to build the CFG graph.

Basic Instruction blocks encompass regular bytecode instructions, including the decision and branching instructions that can influence control flow – namely the sub-types “if”, “goto”, “jsr”, “switch”, “return”, “ret”, “throw”, “sumthrow” and “exit”. They are represented in the CFG by Basic Instruction nodes. Call blocks represent bytecode instructions that cause control flow to be transferred to another method; they contain the high-level information needed to identify the method being called, and are represented in the CFG by Call nodes. In our example, bytecode instructions are grouped in accordance to Table 1.

Table 1. Mapping between bytecode instructions, basic instruction blocks, basic intruction block subtypes, and node numbers in the CFG depicted in Figure 3.

initial bc inst	final bc inst	node type	node subtype	node number
0	1	Call		2
4	5	Basic	If	4
8	17	Basic	Throw	5
18	19	Call		6
22	25	Basic	If	8
28	29	Call		9
32	39	Basic	If	11
38	47	Basic	Throw	12
48	55	Call		13
58	63	Basic	Return	15

Additionally, other types of nodes which represent virtual operations are defined: Entry nodes, Exit nodes, and Return nodes. These virtual nodes encompass no bytecode instructions; they are used to represent certain control flow hypothesis.

As mentioned above, Call blocks transfer control flow to the CFG of another method; the method called, in turn, can return normally or with an exception. In order to differentiate these situations, Return nodes are created. They follow Call nodes, and are transversed when the called method returns regularly; if the called method returns with an exception, either the exception is dealt with internally or control flow jumps to an Exit node that causes the method to return with an exception itself.

Exit nodes follow other nodes that can cause the method to return; a different Exit node is created for each return scenario – including “return” and “throws” instructions, and method call instructions that may return an exception.

Entry nodes identify the starting point of the CFG. They simply indicate the method’s entry point; there’s only one Entry block node per method.

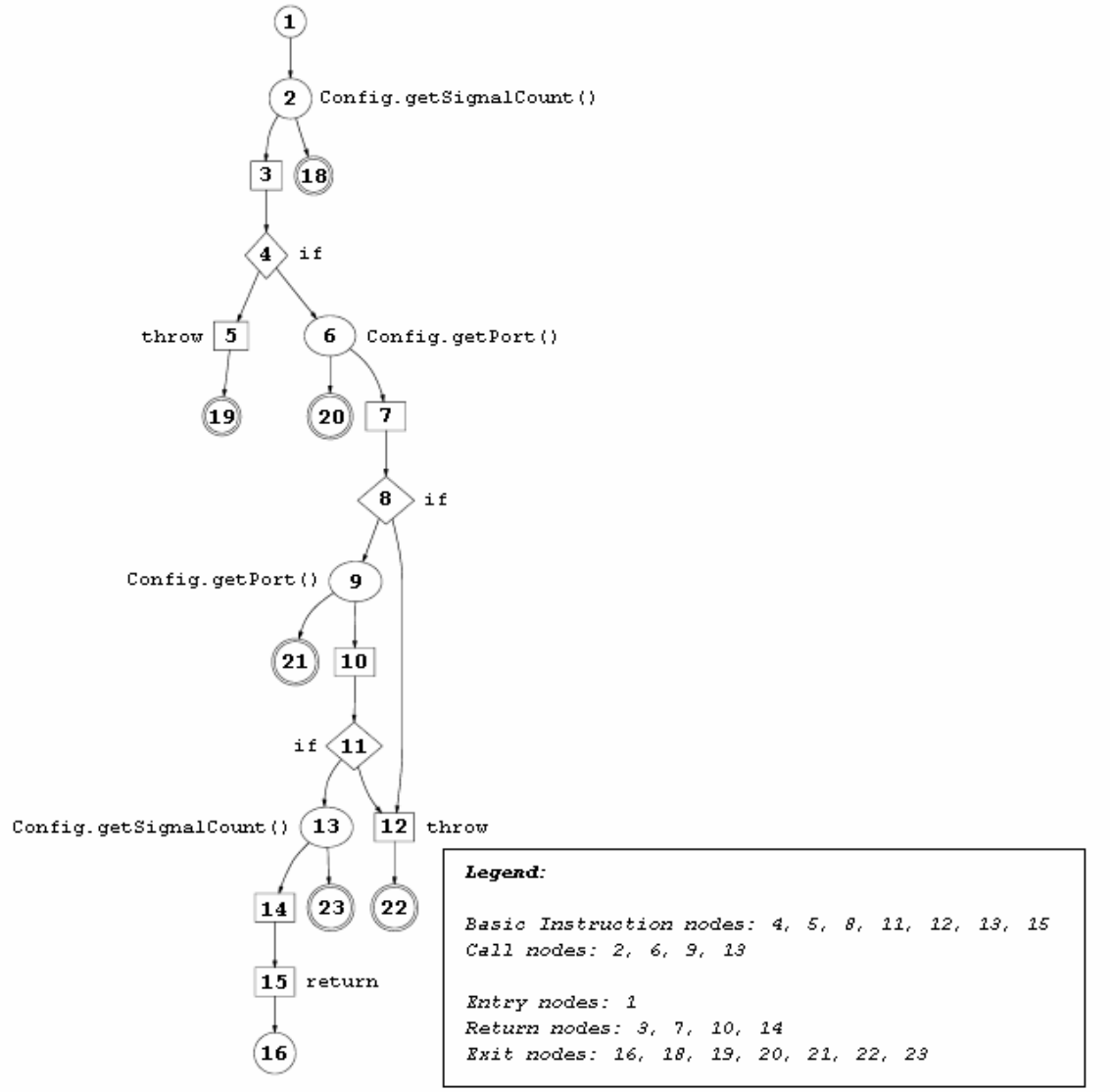

Figure 3. Control-Flow Graph for the method `Controller.reconfigure(Config)`.

The CFG generated for the method `Controller.reconfigure(Config)` is depicted of Figure 3. A brief overview follows: node #1 is the Entry node; it is connected by a directed edge to Call node #2, which transfers control flow to the `Config.getSignalCount()` method; if `Config.getSignalCount()` method returns regularly, Return node #4 is next; if it throws an exception, Exit node #18 is transversed and `Controller.reconfigure(Config)` returns with an exception itself.

4.3. Test Set Evaluation and Optimization

Using the CFG built with basis on the test object's bytecode, it is possible to evaluate thoroughness of the test set with basis on the quality criteria proposed by (Vincenzi, Maldonado et al. 2005; Vincenzi, Delamaro et al. 2006); moreover, it is possible to optimize the test set by employing the evolutionary search techniques proposed by (Wegener, Baresel et al. 2001; Wappler and Wegener 2006a; Wappler and Wegener 2006b). Both methodologies were introduced in the Background section.

In order to access the thoroughness of the testing process according to the all-nodes and all-edges criteria, the test object's class files must be instrumented for Basic block and Call block dispatch, in accordance to the CFG defined as the underlying representation. Dynamic analysis is performed by executing the instrumented test object using each test case of a given test set as input; the trace files produced must then be analyzed for the coverage metrics to be calculated.

```
public void testReconfigure() throws Exception
{
   System.out.println("reconfigure");

   int expected = 8000;

   Config cfg = new Config(9999);
   cfg.setPort(expected);
   Controller instance = new Controller();
   instance.reconfigure(cfg);

   int actual = cfg.getPort();

   assertEquals(expected, actual);
}
```

Figure 4. Sample test case for method `Controller.reconfigure(Config)`

Exception-independent control-flow analysis implies the coverage of the bytecode instructions represented by Basic Instruction nodes and Call nodes (in the case of all-edges$_{ei}$ criterion, by exercising all available branches). Trace files for the execution of the instrumented test object defined in Figure 2 using the test case depicted in Figure 4 yield the transversal of nodes #2, #4, #6, #8, #13 and #15 – i.e. 66% all-nodes$_{ei}$ coverage (nodes #5 and #12 aren't exercised) and 90% all-edges$_{ei}$ coverage (edge #11→#12 isn't exercised; edges beginning at virtual nodes aren't considered) is achieved.

When employing exception-dependant coverage criteria, Exit nodes and Basic Instruction nodes of the subtype "jsr" constitute the focus of the analysis. The all-nodes$_{ei}$ criterion implies the coverage of `catch` and `finally` Java blocks; additionally, the all-edges criterion also implies the transversal of all the edges that lead to Exit nodes that follow Call nodes.

Specific fitness functions have to be defined for each coverage criterion; we employ the methodologies proposed by (Wegener, Baresel et al. 2001). Each individual fitness function, depending on the coverage criterion of choice, is defined as follows (discussion includes examples related to the coverage information extracted from the trace files described above):

- all-nodes$_{ei}$: a node-oriented fitness function is used, which allows the search to be guided towards achieving every individual test goal – e.g. test cases that exercise nodes #5 and #12 must be created.
- all-edges$_{ei}$: a node-path-oriented fitness function is used, which allows the search to be guided towards reaching a specific problem node and, from there, following a certain path – e.g. a test case that considers node #11 as the problem node and transverses edge #11→#12 must be created.
- all-nodes$_{ed}$: a node-path-oriented fitness function is used, which allows the coverage of all bytecode instructions that are encompassed in `catch` and `finally`

blocks, and that can be reached through a `jsr` bytecode function. Basic Instruction nodes of the subtype "jsr" are the problem nodes.

- all-edges$_{ed}$: a node-node-oriented fitness function is used. In addition to the nodes encompassed by `catch` and `finally` blocks, test cases must be generated that reach every single Call node and, from there, transverse the Exit node that corresponds to the called methods exceptional return – e.g. test cases must be generated that consider nodes #18 to #23 as individual goals for the evolutionary search.

5. Framework Description

The focus of our tool is on the creation and optimization of a test set that maximizes code coverage. Optimization occurs at the test set level and at the test case level: we aim to generate a set that can help gain confidence in the software under test using white-box metrics, and to generate the shortest sequence for a given test goal.

The process of CFG building, bytecode instrumentation and event tracing is achieved with the aid of Sofya (Kinneer, Dwyer et al. 2006), a dynamic analysis framework developed at the University of Nebraska, USA, that is particularly suited for developing dynamic analysis tools. The Sofya package provides implementations and tools for the construction of various kinds of graphs – most notably CFGs – and native capabilities for dispatching event streams of specified program observations, which include instrumentators, event dispatchers, and event selection filters for semantic and structural event streams. Additionally, it contains tools to perform various analyses using the outputs generated by its components (statistics, coverage reports, …) and to visualize the trace files produced by the executions of instrumented programs.

In the context of our tool, Sofya is employed to instrument classes for structural event dispatch. Basic Block instrumentation enables the observations of the virtual Entry, Exit and Return blocks, Call blocks and Basic Instruction blocks transversed during a given program execution. Our tool automatically executes instrumented programs, and compares the trace files produced to the statically generated CFGs in order to compute the fitness function.

For evolving the set of test cases, the Evolutionary Computation in Java (ECJ) package (Luke, Panait et al. 2007) is used in a similar fashion to the one proposed in (Wappler and Wegener 2006a; Wappler and Wegener 2006b). ECJ is a research package, developed at the George Mason's University, USA, that incorporates several Universal Evolutionary Algorithms, and includes built-in support for Set-Based Strongly-Typed Genetic Programming. It is highly flexible, having nearly all classes and their settings being dynamically determined at runtime by user provided parameter files.

Parameter files containing all the constraints defined by the function set are automatically generated by our tool: firstly, the Test Cluster and the Type Set for the Class Under Test are extracted by means of the Java Reflection API; then, the Extended Method Call Dependence Graph (EMCDG) is computed, and a Function Set for each of the public methods is derived; finally, ECJ parameter files are automatically generated for each of the function sets produced.

jUnit is used as the front-end for our tool, as it constitutes both the starting and ending points of the software testing process: the initial population of test cases can optionally be derived from those defined by the user using jUnit (the initial population can also be created automatically), and the generated test programs can be transformed into test classes that can be loaded into the jUnit framework. The major usage scenario is the generation of test cases that complete a test set in order to maximize code coverage. The rationale for minimizing the length of the method call sequence of test cases is that of simplifying the user's task of defining assertions.

6. Conclusions and Future Work

This paper presents the rationale and introduces the methodology for generating and optimizing test sets with basis on metrics derived from the dynamic analysis of the test object's Java bytecode. A Control-Flow Graph is used as the underlying model for program representation, and it is build solely with basis on the high-level information extracted from the Java bytecode of the test object. Bytecode instructions are grouped into a smaller set of Basic Instruction and Call blocks with the intention of easing the representation of the test object's control flow, and additional virtual nodes are defined to facilitate the dynamic analysis phase. The methodology for evaluating the test set includes instrumenting the bytecode for basic block analysis and structural event dispatch, and executing the instrumented test object using the generated test cases as inputs, with the intention of collecting trace files with which to derive coverage metrics. Methodologies for defining fitness functions in order to achieve the particular criteria-related test goals are introduced. A general overview on how our automated software testing tool is integrated is given.

Evolutionary testing is an emerging methodology for automatically generating high quality test data. Future work includes performing a case-study in a real development context in order to demonstrate the usefulness and applicability of the methodology and experiment different approaches to the evolutionary paradigm employed. Namely, we aim to fine-tune the fitness functions employed for working at the bytecode level. Further research must also be made on the topic facilitating the user's task of defining assertions for the generated test cases (e.g. by minimizing the length of method call sequence of test cases) and on the possibility of using distinct strong-typing mechanisms for the definition of the constraint imposed by the object-oriented paradigm.

7. References

Cheon, Y., M. Y. Kim, et al. (2005). A complete automation of unit testing for Java programs. Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05). Las Vegas, Nevada, USA, CSREA Press: 290-295.

Haggar, P. (2001, 2001/07/01). "Java bytecode: Understanding bytecode makes you a better programmer " IBM developerWorks Retrieved 2007/04/01, from http://www-128.ibm.com/developerworks/ibm/library/it-haggar_bytecode/.

Kinneer, A., M. Dwyer, et al. (2006). Sofya: A Flexible Framework for Development of Dynamic Program Analyses for Java Software, Department of Computer Science and Engineering, University of Nebraska - Lincoln.

Lindholm, T. and F. Yellin (1999). The Java virtual machine specification. Harlow, Addison-Wesley.

Liu, X., B. Wang, et al. (2005). Evolutionary search in the context of object-oriented programs. MIC2005: The Sixth Metaheuristics International Conference, Vienna, Austria.

Luke, S., L. Panait, et al. (2007). "ECJ 16: A Java evolutionary computation library." from http://www.cs.gmu.edu/~eclab/projects/ecj/.

Mantere, T. and J. T. Alander (2005). "Evolutionary software engineering, a review." Applied Soft Computing 5(3): 315-331.

McMinn, P. (2004). "Search-based software test data generation: a survey." Software Testing, Verification and Reliability 14(2): 105-156.

McMinn, P. and M. Holcombe (2003). The state problem for evolutionary testing. Genetic and Evolutionary Computation Conference, Chicago, USA, Springer-Verlag.

Muller, R. A., C. Lembeck, et al. (2004). A symbolic Java virtual machine for test case generation. Proceedings of IASTED Conference on Software Engineering: 365-371.

Tonella, P. (2004). Evolutionary testing of classes. ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. Boston, Massachusetts, USA, ACM Press: 119-128.

Tracey, N., J. Clark, et al. (2002). A search-based automated test-data generation framework for safety-critical systems, Springer-Verlag New York, Inc.

Vincenzi, A. M. R., M. E. Delamaro, et al. (2006). "Establishing structural testing criteria for Java bytecode." Software Practice and Experience 36(14): 1513-1541.

Vincenzi, A. M. R., J. C. Maldonado, et al. (2005). "Coverage testing of Java programs and components." Special issue on new software composition concepts 56(1-2): 211-230.

Wappler, S. and F. Lammermann (2005). Using evolutionary algorithms for the unit testing of object-oriented software. GECCO '05: Proceedings of the 2005 conference on genetic and evolutionary computation, ACM Press: 1053-1060.

Wappler, S. and J. Wegener (2006a). Evolutionary Unit Testing Of Object-Oriented Software Using A Hybrid Evolutionary Algorithm. Proceedings of the 2006 IEEE Congress on Evolutionary Computation. Vancouver, IEEE Press: 3193-3200.

Wappler, S. and J. Wegener (2006b). Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation. Seattle, Washington, USA, ACM Press: 1925-1932.

Wegener, J., A. Baresel, et al. (2001). "Evolutionary test environment for automatic structural testing." Information & Software Technology 43(14): 841-854.