A JVM-based Testing Harness for Improving Component Testability

Weifeng Xu
Department of Computer Science
Bowie State University, Bowie, USA
frank.w.xu@gmail.com

Omar El Ariss
Department of Computer Science & Mathematical Science,
Penn State Harrisburg, PA, USA
oue1@psu.edu

***Abstract*— Software testing is a critical activity in increasing our confidence of a system under test and improving its quality. The key idea for testing a software application is to minimize the number of faults found in the system. The higher the testability of software, the better our chances to reveal these faults. We introduce a new type of testing harness called GannonJVM that improves the testability of software components. GannonJVM enhances the Java Virtual Machine (JVM) with a predicate analyzer and a bytecode interpreter. Our automated test framework is able to extract and visualize paths from the control flow graph of a given component. We also observe and analyze the predicates in a given a path during runtime.**

Keywords*— *Software testing, test harness, bytecode, testing tool, visualization.

I. INTRODUCTION

Testability is a measure of how complex it is to test a software application. The lower the testability of a software, the lower the quality of the generated test cases. On the other hand, the higher the testability of software the better the test cases are. Two major factors that contribute to the testability of software are controllability and observability. Controllability is the ability to access, and the ease of testing the features and functionalities of an application. Observability is the extent to which the output or the observable states of the system assist in the verification of the test results.

A test harness typically refers to a testing framework. It is used to execute test cases and to check whether the actual results match the expected ones. Junit is a popular test harness. For example, the assertion `assertEquals("Isosceles", new Trianlge (7,7,6).getTriType())` in JUnit checks whether a triangle object reports the correct triangle type, which is an isosceles for the given three side values: 7, 7 and 6. Although JUnit and other similar unit testing frameworks are capable of executing test cases and checking their results automatically, they usually do not focus on improving the testability (i.e., testing observability and controllability) of software components. For example, JUnit does not provide any feedback that helps the testers to redesign test inputs. It also does not provide useful runtime states for debugging when the test case fails.

To deal with these limitations, we introduce a novel approach to develop a test harness directly on top of the Java Virtual Machine (JVM). We call this approach GannonJVM. Its aim is to improve the testability of Unit Under Test (UUT). Fig. 1 shows the three main components of GannonJVM: (1) a predicate analyzer that is designed to improve the testing observability of UUT. The analyzer defines how the internal states of UUT are monitored and inferred by the knowledge of its external input. It checks the predicate values and monitors their execution paths in terms of the test inputs. (2) a Bytecode interpreter that improves the testing controllability of UUT. The interpreter defines how to stabilize an execution path based on its observation. It is used to adjust the original test input (i.e., seed value) to create additional input that forces the UUT to execute a designated path. (3) Control Flow Graph (CFG) visualizer. It is responsible for automatically determining the layout of a CFG.

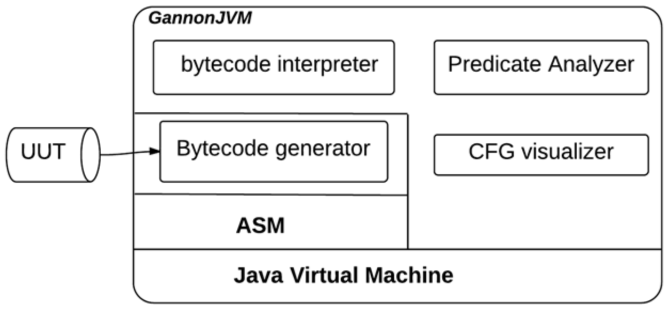


Fig. 1. Overview of the test harness

The rest of this paper is organized as follows: Section II introduces the basic concepts through a running example. Section III and IV describe the predicate analyzer and inference engine. Section V describes the CFG visualizer. Section VI reviews the related work. Section VII concludes the paper.

II. A RUNNING EXAMPLE

We use the classical *Triangle Problem* [1] as a running example to illustrate the test harness with embedded features for observability and controllability to generate test input without fitness functions. Given three positive integers that represent the

(DOI Reference Number: 10.18293/SEKE2015-043)

lengths of the three sides of a triangle, the *Triangle* program reports the triangle type: *Equilateral (type 1), Isosceles (type 2), Scalene (type 3),* or *NotATriangle (type 4)*. The source code for this problem is shown below:

```
int getTriType (a,b c) {
  if ((a<b+c) && (b<a+c) && (c<a+b) ){
    if (a==b && b==c)    return 1;
    else if (a!=b && a!=c &&b!=c) return 3;
    else  return 2;}
  else  return 4;}
```

Java bytecode is a stack-oriented language, which pops data (operands) from the top of the stack and pushes data back on the top of the stack. The stack is commonly referred to as an operand stack [2]. For example, the bytecode instruction `iadd` pops two integer values from the stack and pushes their sum back to the stack. Two integer values are pre-loaded from a *local variable table* using two `iload` instructions. To facilitate the discussion, the bytecode of *Triangle* instructions are divided (see dashed lines) into 20 blocks shown below [3].

```
1. iload 1          | 16. iload 1          | 28. iload 3
2. iload 2          | ------------[7]      | 29. if_icmpeq 35
3. iload 3          | 17. iload 2          | ------------[15]
4. iadd             | 18.if_icmpne 24      | 30. iload 2
------------[1]     | ------------[8]      | ------------[16]
5. if_icmpge 37     | 19.  iload 2         | 31. iload 3
------------[2]     | 20.  iload 3         | 32. if_icmpeq 35
6. iload 2          | ------------[9]      | ------------[17]
------------[3]     | 21. if_icmpne 24     | 33. iconst_2
7. iload 1          | ------------[10]     | ------------[18]
8. iload 3          | 22. iconst_1         | 34. ireturn
9. iadd             | -----------[11]      | 35. iconst_3
10.if_icmpge 37     | 23. ireturn          | ------------[19]
------------[4]     | 24. iload 1          | 36. ireturn
11. iload 3         | ------------[12]     | 37. iconst_4
------------[5]     | 25. iload 2          | 38. ireturn
12. iload 1         | 26.if_icmpeq 35      | ------------[20]
13. iload 2         | -----------[13]      |
14. iadd            | 27. iload 1          |
15.if_icmpge 37     | ------------[14]     |
-----------[6]      |                      |
```

Bytecode instructions have unique properties. First, they have an implicit effect on the stack as each instruction has no explicit named operands. For example, `iadd` (instruction 4) in block 1 does not specify the two operands that will be fetched for integer addition. These values are determined by `iload 2` and `iload 3` (instructions 2 and 3) as they are the top two values on the current operand stack. Note that the operand of `iload` points to the index of the local variable table [2]. The local variable table contains bytecode instructions and input parameters after initializing the method invocation. For example when the method is invoked, the three input variables `a`, `b`, and `c` of the triangle program are stored in the first three spots of the local variable table. During execution, `iload 1`, `iload 2`, and `iload 3` push the values stored in the indices *1*, *2*, and *3* to the operand stack. Second, predicates with multi-conditions in Java source code are represented by multi-level conditions in bytecode. For example, the multi-condition `(a<b+c)&&(b<a+c)&& (c<a+b)` in the *Triangle* source code is decomposed into three block sets, i.e., blocks 1 and 2, blocks 3 and 4, as well as blocks 5 and 6. Blocks 2, 4, and 6 are three identical `if_icmpge` statements. They are depicted in Fig. 2 by the CFG of *Triangle* bytecode. This property facilitates observability and controllability by decomposing component conditions into several simple conditions.

A path that is generated from a CFG consists of a sequence of blocks. For example, p1=[1]→[2]→[3]→[4]→[5]→[6]→[7]→[8]→[9]→[10]→[11] is a path for testing if a triangle is *equilateral*. To execute all the blocks in p1, the values that participate in the evaluation of predicates [2] [4] [6] [8] and [10] need to be adjusted so that the predicates produce the desired outcomes to reach the last block. Thus, the desired outcome for each predicate in p1 should be achieved as p1: [1]→[2] $\xrightarrow{F}$[3]→[4]$\xrightarrow{F}$[5]→[6] $\xrightarrow{F}$[7] →[8] $\xrightarrow{F}$[9]→[10] $\xrightarrow{F}$[11], where F (False) is the expected outcome of the corresponding predicate. Such a path is called a **tagged path**. A tagged path is a sequence of edges that have at least one **tagged edge**, where a tagged edge is defined as $v\overset{o}{-}{>}u$. The variable *v* is the source block that represents a predicate in a statement, *o* is a **tagged value** for *v*, which represents the desired outcome of *v (i.e., true* or *false)*, and *u* is the reachable block if the assertion `asserEquals(`*o*`, runtime(`*v*`))` returns true.

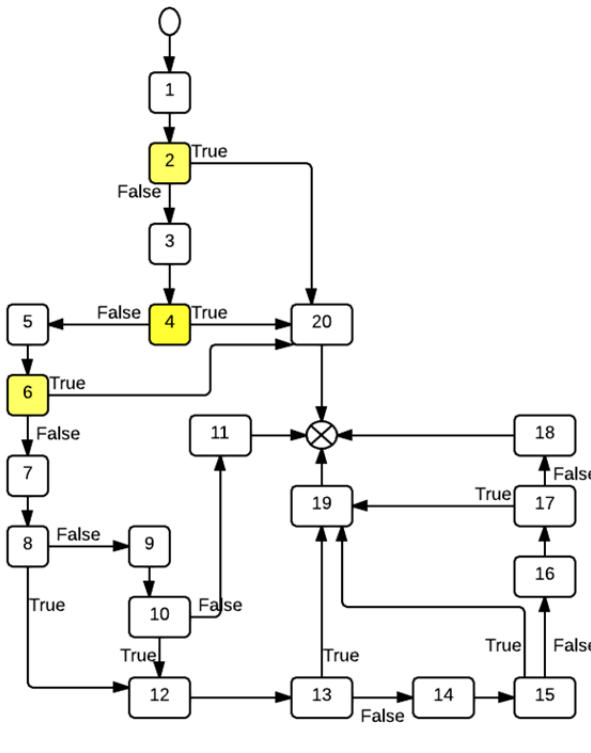


Fig. 2. CFG of the Triangle Problem in bytecode

Table 1 shows some representative tagged paths of the Triangle program based on decision coverage. In tagged path p12, the tagged edge $[2] \xrightarrow{T} [20]$ indicates that block 2 is a predicate `if_icmpge` and its outcome must be true in order to reach block 20.

Table 1. Tagged paths for Triangle problem

Goal	ID	Path
Equilateral	1	$[1]\rightarrow[2]\xrightarrow{F}[3]\rightarrow[4]\xrightarrow{F}[5]\rightarrow[6]\xrightarrow{F}[7]\rightarrow[8]\xrightarrow{F}[9]\rightarrow[10]\xrightarrow{F}[11]$
Isosceles	2	$[1]\rightarrow[2]\xrightarrow{F}[3]\rightarrow[4]\xrightarrow{F}[5]\rightarrow[6]\xrightarrow{F}[7]\rightarrow[8]\xrightarrow{F}[9]\rightarrow[10]\xrightarrow{T}[11]\rightarrow[12]\rightarrow[13]\xrightarrow{T}[19]$
	…	
Scalene	8	$[1]\rightarrow[2]\xrightarrow{F}[3]\rightarrow[4]\xrightarrow{F}[5]\rightarrow[6]\xrightarrow{F}[7]\rightarrow[8]\xrightarrow{T}[12]\rightarrow[13]\xrightarrow{F}[14]\rightarrow[15]\xrightarrow{F}[16]\rightarrow[17]\xrightarrow{F}[18]$
	..	
NotATriangle	..	
	12	$[1]\rightarrow[2]\xrightarrow{T}[20]$

III. PREDICATE ANALYZER

The predicate analyzer is designed to improve the observability of UUT. It examines bytecode instructions to discover relationships between input variables and variables used in predicates. Discovering relationships relies on variable binding and variable dependency analysis.

The process of binding explicit variables to bytecode instructions is called variable binding. As bytecode instructions have an implicit effect on the evaluation stack, an effective approach is to use instruction tree unit (ITU) as an intermediate representation of instructions. Each ITU is a binary tree, which consists of three nodes, one parent node and two child nodes, as well as an operator (i.e., the opcode of the instruction) between the two children. The child nodes are the explicit named operands. The root is a named intermediate result of the operation. An ITU can be simply represented by a four-tuple *(opcode, root, leftNode, rightNode).* One of the essential characteristics is that ITUs are restricted to the least number of operands (2 in most cases, such as for arithmetic and logic), and these operands must either be constants or locals. For example, for a given Java expression statement `x=a+b+c`, the corresponding two arithmetic ITUs are shown in Fig. 3. Local variables `i0`, `i1`, `i2` are stored in the local variable table and correspond to the variables `a`, `b`, `c` and `x` in the given Java statement. Variables with a "`$`" sign are intermediate local variables, e.g., `$i4` is an intermediate variable for holding the value of `i0 + i1` and `$i5` holds the result of `$i4 + i2`. `iadd` is the opcode of the instruction `iadd #index`, where `#index` is the index of the local variable table.

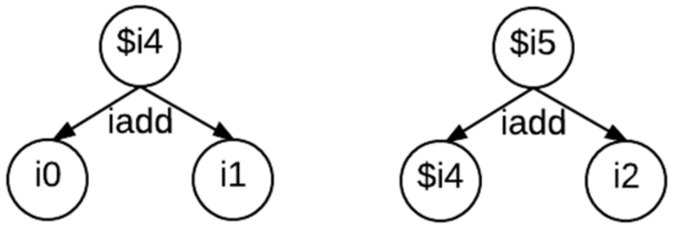


Fig. 3. Two ITUs of Java expression statement `x=a+b+c`

Variable binding is a dynamic process, which builds the ITUs along with the execution of bytecode. We utilize an additional stack, called *variable binding stack,* and a variable table, called *variable binding table,* to bind variables to instructions. We gave them these names to distinguish them from the operand stack and the local variable table specified by the JVM specification. The *variable binding stack* and *variable binding table* work very similar to the JVM operand stack and the local variable table except that 1) the *variable binding stack* and *variable binding table* store the names of the bytecode intermediate variables instead of the operands for tracking intermediate variables, and 2) each element of the *variable binding table* also has a reference point to the root of the ITU containing itself.

Variable binding during instruction execution in the JVM works as follows: 1) whenever an instruction pushes a value into the operand stack, and the value is loaded from the local variable table, the index of the value in the local variable table is used as the intermediate local variable name. This index is pushed into the *variable binding stack*. Otherwise, a new generated unique ID is used as the name and is pushed into the *variable binding stack.* 2) whenever an instruction pops a value from the operand stack, the top of the *variable binding stack* is removed as well. The popped intermediate local variable names are used for constructing the ITUs. Note that for the purpose of dependency analysis, we build ITUs only for instructions that produce an effect on the operand stack and are influenced by the effect, i.e., instructions that produce and use intermediate variables. Therefore, ITUs are categorized into two groups: expression ITU and predicate ITU. Expression ITUs are built from expression instructions [4] producing intermediate variables, including load, arithmetic, and logic instructions. Predicate ITUs are built from predicate instructions using the intermediate variable to compute the tagged values, including all if_* Instructions. The algorithm can be applied for binding other instructions. Fig. 4 shows the variable binding results (i.e., the two ITUs) for the tagged path p12: $[1]\rightarrow[2]\xrightarrow{T}[20]$ in Table 1. The block list is a variable binding table. The first three variables, `i0`, `i1`, and `i2`, are the names of the input parameters. `$i10` and `$i11` are intermediate variables pointing to the root of the two ITUs shown in Fig. 4. The letter "`i`" is added before the generated ID as part of variable name for readability.

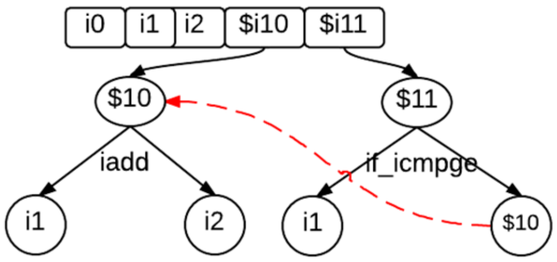


Fig. 4. Variable binding for path P12

Variable dependency analysis is the process of backtracking input variables for a given intermediate bytecode variables for making the assertion `asserEquals(o, runtime(v))` to be true. Again, considering the simple tagged path p12: the goal is to find a test input to execute this path (i.e., find a triangle type of "*NotATriangle*"). As $[2]\xrightarrow{T}[20]$ is the only tagged edge, the path will be covered if a test input forces the constraint in the instruction `if_icmpge` to be *true* (statement 5 is the only instruction in block [2]). The predicate ITU `(if_icmpge, $i11, $i1, $i10)` indicates that to generate a test input to cover p12, however, we need to determine the input variables that are associated with `$i10`. The association will allow the proposed system to backtrack the input variables so that they can be adjusted to meet the constraint. It is not difficult to see that `$i10` (shown in the expression ITU `iadd` on the left of Fig. 4) is associated with input variables `i1` and `i2` by backtracking `$i10` in the predicate ITU on the right. Variable dependency can be graphically captured using a Variable Dependency Tree (VDT). A VDT consists of a set of ITUs, where the root and each intermediate node are intermediate variables, and all leaves are the bytecode input variables. The algorithm below describes the procedure for building VDTs from ITUs. The algorithm recursively expands child nodes containing intermediate variables with ITUs. The red dashed line shown in Fig. 4 indicates a backtracking relation of `$10`.

```
Algorithm: Building VDTs
Inputs: VBT: A variable binding table
Outputs: VDT: A variable dependency tree
procedure buildVDTs(VBT)
  for each element  E of VBT
    (opeCode, root, leftNode, rightNode) ⟵
  E.getITU()
    if leftNode/rightNode of the ITU containing
    intermediate variable
      newITU ⟵ find a new ITU based on leftNode or
      rightNode
      Point from leftNode/rightNode to newITU
    end if
  end for
end procedure
```

IV. BYTECODE INTERPRETER

Bytecode interpreter aims to improve testing controllability of UUT, i.e., how to control the predicate evaluation results to force a given path to be executed at run-time. Note that the evaluation results are determined by the input, where the rule-based inference engine provides input changing guidelines.

Bytecode interpreter controls the order of which bytecode instruction will be fetched and executed. It reads each bytecode instruction and returns the evaluation result. It mainly consists of a program counter, which points to the next instruction to be fetched and executed, a local variable table, and an operand stack. In addition, a Java stack is needed for method invocations. Each element of the Java stack is a Java frame, which stores execution status. To make the interpreter more flexible, we utilize a factory design pattern to encapsulate instruction creation and a strategy pattern to encapsulate the execution algorithm in each instruction. A snapshot of the implementation of `BIFicmpge` instruction is shown below. The execution method implements the abstract method defined in the `Instruction` class. This comparison instruction pops two values from the operand stack and returns the predicate result. It is worth noting that the bytecode input parameters are stored at the beginning of the local variable table. They will be fetched for UUT interpretation. It is not difficult to overwrite them with new generated input in order to make the input generating process automatic. Along with the predicate analyzer and rule inference engine, this overwriting mechanism makes the UUT running until a given path is executed.

```
public class IFicmpge extends Instruction {
@Override
public Object execute(JavaFrame  frame) {
    Stack<Integer> opStack =
frame.getOperandStack();
    Integer rightValue = (Integer) opStack.pop();
    Integer leftValue = (Integer) opStack.pop();
    boolean result=rightValue>= leftValue;
     return result;
}
```

The Bytecode interpreter then collaborates with the *Bytecode generator***.** Compiled Java class files are in the form of hexadecimal. Therefore, ASM [5] is utilized to convert hexadecimal numbers to readable bytecode instructions. ASM is a very small and very fast Java bytecode manipulation framework supported by Open Solutions Alliance.

V. CONTROL FLOW GRAPH VISUALIZER

A directed graph G = {V, E} consists of two types of elements `V` and `E`, where `V` is a set of vertices and `E` is a set of edges. A Control Flow Graph (CFG) is a graph with some special vertices and edges: 1) it has source and sink vertices and 2) it consists of loops and jumps. Control Flow Graph (CFG) visualizer is responsible for determining the layout of a CFG automatically. CFG visualizer needs to solve three challenges: 1) how to calculate the layout of graph if we treat CFG is a general type of graph, 2) how to handle with two special vertices, i.e., source and sink, and 3) how to determine two special edges of CFG, i.e., loops and jumps.

A. *Visualizing CFG as A Normal Graph*

Force-directed algorithms are the most flexible and popular algorithms for calculating layouts of simple undirected graphs. These algorithms calculate the layout of a graph using only information contained within the structure of the graph itself. For a given directed graph `G = {V, E}`, a force-directed algorithm models edges as springs and vertices as charged particles. Springs represent attractive forces based on Hooke's law, which are used to attract pairs of connected vertices towards each other. Charged particles represent repulsive forces based on Coulomb's law, which are used to separate all pair of vertices. Force is represented as a vector, which includes a magnitude and direction. In a force-directed algorithm, we start with assigning a random position for each vertex. Then each vertex applies the attractive and repulsive forces. This will cause the vertex to move to a new position. The calculating and moving activities repeat until the graph reaches equilibrium states. In equilibrium states for a given graph, edges tend to have uniform length because of the spring forces, and nodes that are not connected by an edge tend to be drawn further apart because of the electrical repulsion.

Fig. 5 shows the automated layout calculation using the attractive and repulsive forces on the Triangle problem CFG. Vertices in Fig. 5 (a) are assigned random positions. Fig. 5 (b) shows the equilibrium states of the CFG.

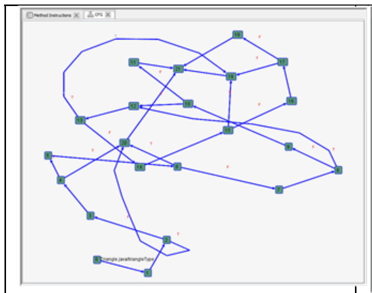
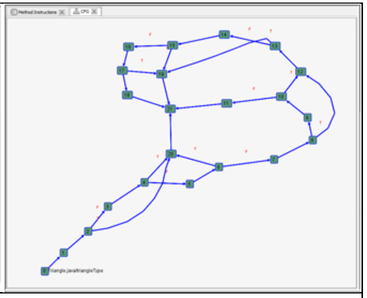
	
(a) Initial state	(b) Equilibrium states

Fig. 5. Triangle CFG layout with two forces

The force-directed algorithm is defined below and is based on Eades' idea [6]:

```
Algorithm SPRING(G: graph)
  Place vertices of G in random locations
  Repeat M times
  Calculate the force $\vec{F}(v)$ on each vertex
  Move the vertex based on force on vertex
  Draw graph on screen
End of Algorithm
```

The force $\vec{F}$ (v) is defined as:

$$\vec{F}(v) = \sum_{(u,v)\in V\times V} \vec{H}_{uv} + \sum_{(u,v)\in E} \vec{C}_{uv} \quad (1)$$

Where $\vec{H}_{uv}$ represents the attractive force between two connected vertices, *u* and *v*, calculated based on Hooke's law. $\vec{C}_{uv}$ represents the repulsive force between vertices *u* and *v*, and is calculated based on Coulomb's law.

B. *Positioning Source and Sink Vertices*

The control flow graph $G(f) = \{V, E, v_{in}, v_{out}\}$ of a function `f` has two additional vertices, source and sink vertices, referred as v_{in} and v_{out}, respectively. A source vertex is a vertex with indegree zero, while a sink vertex is a vertex with outdegree zero. The control flow graph of an empty function, i.e., a function without any statements consists of $V = \{v_{in}, v_{out}\}$ and $E = \{(v_{in}, v_{out})\}$.

Unlike the layout solution shown in Fig. 5, traditionally, all vertices of a CFG are arranged in the form of top-to-bottom where v_{in} and v_{out} are placed on the top and bottom positions, respectively. In order to rearrange v_{in} and v_{out} in Fig. 5, the third force, named *Earth Gravitational Force*, is added to formula (1). The gravity of Earth, denoted as $\vec{T}$, refers to the acceleration that the Earth imparts to objects on or near its surface.

The Earth Gravitational Force is defined as:

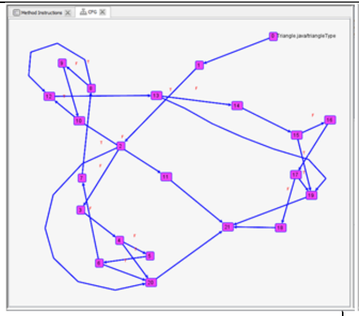
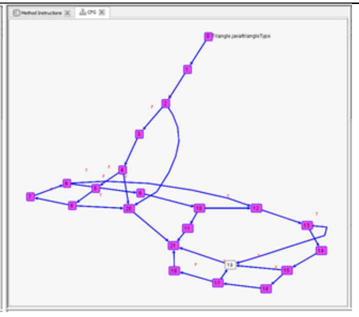
$$\vec{T}(v) = mg \quad (1)$$

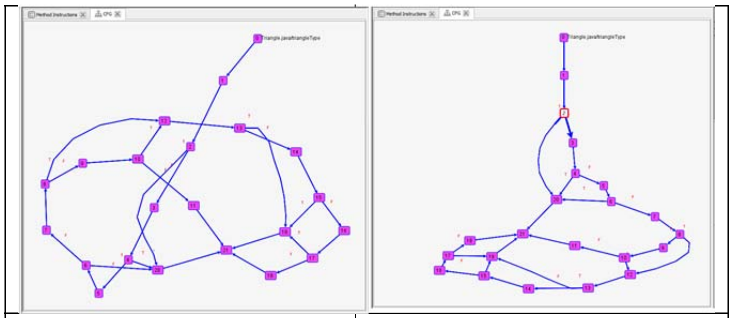
Where, `m` is the mass of the vertex and `g` is the gravitational content.

The new formula for handling source and sink vertices is now defined as:

$$\vec{F}(v) = \sum_{(u,v)\in V\times V} \vec{H}_{uv} + \sum_{(u,v)\in E} \vec{C}_{uv} + \sum_{(u)\in E} \vec{T}_u \quad (2)$$

Fig. 6 shows the automated calculated *Triangle* CFG layout with the additional Earth gravitational force. Fig. 6 (a) (b) (c) (d) illustrates the evaluations of the CFG layout.

	
(a) Initial state	(c) Iteration 20



(b) Iteration 10 | (d) Equilibrium states

Fig. 6. Triangle CFG layout with Earth gravitational force

C. Positioning Loops and Jumps Edges

There are two types of special edges, loops and jumps (i.e., loop and if-else statements) in a CFG. For example, v_2 in Fig. 6 is a predicate node containing an if-else statement. Without an appropriate positioning algorithm, the edge $(v_2,\ v_{20})$ will be a straight line. Positioning such special edges need (1) Identifying dominator relationships: In a CFG graph, a vertex v dominates another node w if and only if every directed path from v_{in} to w in the CFG contains v. The dominators of node w is defined as $dom\ (w) = \{v \mid v \text{ dominates } w\}$. For example, $dom\ (v_{20}) = \{v_0, v_1, v_2\}$. (2) Identifying special edges: The node $v_{shortest} = v \in dom\ (w)$ has the shortest path from v to w, where v is the start node and w is the end node, i.e., the special edge is defined as $(v_{shortest},\ w)$, and (3) Adding invisible vertices to special edges: The number of invisible vertices equals to the number of vertices from $v_{shortest}$ to w.

VI. Related Work

Various testing harnesses have been explored to monitor the runtime state of UUT. These tools mainly fall into two categories: aspect-oriented approaches and symbolic execution based approaches. MOP [7] is a Monitoring-Oriented Programming (MOP) framework, which automatically generates monitors from the specified properties and then integrates them together with the user-defined code into the original system. In the implementation, parametric specifications are translated into AspectJ [8] code, and then weaved into the application using off-the-shelf AspectJ compilers. Tracematches [9] is another aspect-oriented trace-matching tool to observe the execution of a base program; when certain actions occur, the aspect runs some extra code of its own. Java Pathfinder (JPF) [10][11] is a system to verify executable Java bytecode programs. It is based on symbolic execution for test case generation. The core of JPF is a Java Virtual Machine that is also implemented in Java. JPF executes normal Java bytecode programs and can store, match and restore program states. KLOVER [12] is similar to JPF. It executes and monitors the states of running C++ program in the form of LLVM bytecode. GannonJVM implements the features of testing observability and controllability, which monitors, interpreters and controls the Java bytecode instructions directly using a stack-based approach.

VII. Conclusion

This paper presents a novel approach to embed two testability features, including testing observation and testing control features. We also introduce a CFG visualization in Java Virtual Machine (JVM) for building a new testing harness to facilitate software testing. The implementation of GannonJVM, a demo video, and the *triangle* example are publicly available[1].

VIII. Acknowledgment

We thank Syed Aqeel Raza and Bader Aldawsari for assistance with the implementation of the system.

References

[1] P. C. Jorgensen, Software Testing: A Craftman's Approach, 3rd ed., Auerbach Publications, 2008.

[2] T. Lindholm, F. Yellin, G. Bracha and A. Buckley, Java Virtual Machine Specification, Java SE 7 Edition, Boston, USA: Addison-Wesley Professional, 2013.

[3] J. Zhao, "Analyzing Control Flow in Java Bytecode," in *16th Conference of Japan Society for Software Science and Technology*, Japan, 1999.

[4] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying Java Bytecode for Analyses and Transformations," Sable Research Group, School of Computer Science, McGill University, Montreal, Canada, 1998.

[5] O. Consortium, "ASM," [Online]. Available: http://asm.ow2.org/. [Accessed 23 08 2013].

[6] P. Eades, "A heuristic for Graph Drawing," *Congressus Numerantium*, vol. 160, no. 42, p. 149, 1984.

[7] F. Chen and G. Rosu, "MOP: An Efficient and Generic Runtime Verification Framework," in *Object-Oriented Programming, Systems, Languages & Applications*, Nashville, Tennessee, 2007.

[8] "AspectJ," Eclipse Foundation, [Online]. Available: http://eclipse.org/aspectj/. [Accessed 10 Sep 2013].

[9] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. d. Moor, D. Sereni, G. Sittampalam and J. Tibble, "Adding Trace Matching with Free Variables to AspectJ," in *Object-Oriented Programming, Systems, Languages & Applications*, San Diego, California, 2005.

[10] N. A. R. Center, "Java Pathfinder," NASA Ames Research Center , [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf. [Accessed 27 1 2014].

[11] C. S. Pasareanu, et al., "Symbolic PathFinder: Integrating Symbolic Execution with Model Checking for Java Bytecode Analysis," *Automated Software Engineering*, vol. 20, no. 3, p. 391-425, 2013.

[12] G. Li, I. Ghosh and S. P. Rajan, "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs," in *23rd International Conference on Computer Aided Verification*, Snowbird, Utah, 2011.

[1] Implementation and example:http://perceval.gannon.edu/xu001/research/GannonJVM/. Source code: git@github.com:Gannon-University/GannonJVM.git. Demo video: https://www.youtube.com/watch?v=Ey4JfVhhHQg.