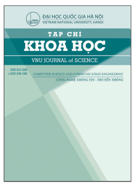
VNU Journal of Science: Comp. Science & Com. Eng., Vol. 40, No. 1 (2024) 18–38

VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce



Original Article

A Control Flow Graph Generation Method for Java Projects

Hoang-Viet Tran[*], Pham Ngoc Hung

VNU University of Engineering and Technology, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam

Abstract: Many software quality assurance methods depend on the control flow graph (CFG) for the analyzing process. However, existing methods for Java projects have not described details of the CFG generation process, which is critical for having a fully automatic analysis method. This paper presents a novel method, named CFG4J, for generating CFG associated with a given Java unit. The generated CFG will be used in many other software quality assurance methods such as test data generation for unit testing, unit debugging, code coverage computation, etc. We have implemented CFG4J in a tool and performed some initial experiments with some common units with potential results. Finally, we give some discussions about the experimental results at the end of the paper.

Keywords: Control flow graph, unit testing, test data generation, Java projects.

1. Introduction

Software quality assurance is one of the most important tasks in software engineering as many aspects of our modern lives depend on software. There are many approaches to addressing software quality assurance problems such as software theorem proving [1–6], model checking [7, 8], software testing [9], etc. Each of these approaches uses different inputs like software models [10–13], software designs, or software source code [14, 15]. While those approaches based on software models mostly depend on experts to create the input models, the other approaches based on software source code can be fully automatic. The reason for this is that from the project source code, we can extract software architecture, models, software dependency graphs, control flow graphs, data flow graphs (DFGs), etc. which can be used as inputs for those software quality assurance methods. Many of these methods use control flow graphs as inputs such as symbolic execution [16, 17], software testing [14, 15, 18], program debugging [19–21], etc.

[*] Corresponding author.
E-mail address: thv@vnu.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.668

Although there are many pieces of research using control flow graphs as their inputs, to our knowledge, there has not been any research describing in detail (i.e., detailed algorithms for practitioners to follow) how to build the corresponding CFG for a Java unit (i.e., a function in a Java class). In 2018, Zambon and Rensink presented a method named recipes for compositional construction of CFG for Java 6 [22]. The method is based on the concept of recipes, which are essentially procedures with atomic behavior. In turn, Those recipes can recursively compose new recipes. However, this method does not show how to build the corresponding CFG for a given Java unit. In 2004, Jang-Wu and Byeong-Mo presented a method to construct CFG by separately computing the normal and exception flows. Their paper has two contributions. First, they show that we can decouple normal flow and exception flow and compute them separately by examining fourteen Java programs. Second, they proposed an analysis that estimates the exception control flow and an exception flow graph that represents the exception control flows. Then, they conclude that we can construct a control flow by merging an exception flow graph onto a normal flow graph [23]. Nevertheless, the paper does not show the details of how to construct the control flow graph from a given Java unit. In 2016, Afshin et al. presented an algorithm to extract flow graphs from Java bytecode, including exceptional control flows [24, 25]. This method does not generate CFG from a given Java source code. There have been many other papers such as the ones of Zaretsky et al. [26] (which generates CFGs and DFGs from assembly code) or Pedro et al. [27] which extracts CFGs from Java bytecode) related to CFG, but they do not mention how to build a CFG from a given Java unit source code.

This paper presents a method, named CFG4J, to generate the corresponding CFG for a given Java unit. We can use the generated CFG as input for other program analysis methods such as test data generation, program debugging, etc. The key idea of CFG4J is to use the abstract syntax tree (AST) of the given Java unit generated by using the Eclipse Java Development Tool[1] (JDT) as input for the CFG generation process. Then, for those statements of the given unit, CFG4J divides them into three main types: sequence statements, condition statements, and loop statements. After that, CFG4J proceeds with the construction of the corresponding CFG of the given unit accordingly. The initial experimental results with some common Java algorithms units show that the time and memory usage of CFG4J is acceptable when being used as input for other program analysis methods.

The rest of the paper is organized as follows. Section 2 shows the overview of the CFG4J, which includes two phases: the generation of AST and the generation process of CFG. This section also presents the method of generating AST for the given Java unit. From the resulting AST, Section 3 shows details of the CFG generation process. Preliminary experiments of CFG4J are shown in Section 4. We discuss the related works to CFG4J in Section 5. Finally, we conclude the paper in Section 6.

2. CFG4J Overview

In this paper, we define the control flow graph as follows.

Definition 1 (Control Follow Graph - CFG)**.** *Given a Java unit, the corresponding CFG is a directed graph $G = (V, E)$, where $V = \{v_0, v_1, .., v_n\}$ is a set of vertices and $E = \{(v_i, v_j)|(v_i, v_j) \in V\}$ is the set of its edges. V represents all basic blocks of the unit while E is a set of directed edges in which each edge (v_i, v_j) represents the program state transition from v_i to v_j.*

[1]https://www.eclipse.org/jdt/

Note 1. *In this paper, we only consider units whose statements are executed sequentially.*

Note 2. *During the CFG generation process (i.e., from Algorithm Gen4Block to Algorithm LinkSpecStmt), when we link two CFG nodes $Node_1$ and $Node_2$ by using the Link, $setTrueNode$, or $setFalseNode$ procedures, we actually create a directed edge from $Node_1$ to $Node_2$. When we create a CFG node from a given AST block or statement, we actually create a CFG node in which the AST block or statement becomes one of its properties (e.g., line 24 of Algorithm Gen4Stmt).*

Given a java unit *u*, CFG4J generates the corresponding CFG for *u* via two main phases: (i) Generate the AST (abstract syntax tree) for *u* and (ii) Generate the corresponding CFG for *u* from the resulting AST. The reason for choosing AST as an input for the CFG generation process is that AST contains all the information we need for generating CFG. In addition, there are libraries that support parsing the source code into AST. This saves us a lot of effort in building the source code parser from scratch. The overview of CFG4J is shown in Figure 1.

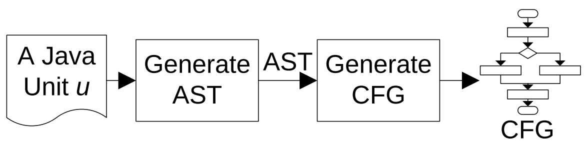


Figure 1. The overview of CFG4J.

- Phase i (Generate AST): From the given Java unit *u*, CFG4J uses the Eclipse Java Development Tools[2] (JDT) to parse *u* and generates the corresponding AST of *u*. For this reason, CFG4J supports those Java versions which are supported by JDT libraries. The latest JDT version supports Java version 5 and above.
- Phase ii (Generate CFG): From the generated AST of Phase i, CFG4J generates the required CFG.

In Phase i, CFG4J generates AST for the given Java unit. For ease of processing, CFG4J employs the JDT libraries. Given a Java project *P*, to test a specific unit *u*, we need to filter all files of *P*, which are Java source files. Then, we read each file as a text string and pass the string to an *ASTParser* (a class in JDT library) instance as the input. Listing 1 shows how a Java source text string can be parsed to find the required function.

The function accepts a source string of a Java file (*sourceCodeFile*) and the function name (*funcName*) as its inputs. After parsing, the function returns the AST node associated with the given function name (*funcName*) inside the source file. At the beginning of the function *parserToAstFuncList*, the list of *ASTNode* representing the corresponding list of functions in the given source file (*AstFuncList*) and an instance of *ASTParser* (*parser*) are initiated as an empty list and a parser of the Java version under consideration (line 3 and 4). Then, the parser creates the AST for the source and stores the result in a compilation unit *cu* (line 5-7). The compilation unit *cu* is visited by using an instance of *ASTVisitor* class (*visitor*) to retrieve the list of units that are not constructors (line 8-24). Finally, the list of AST nodes corresponding to all functions in the given file is checked to find the required function with the name of *funcName* (line 26-32). If the function can be found, the algorithm returns it. Otherwise, the function returns *null* and stops.

After having the AST from Phase i, Phase ii generates the corresponding CFG of the given Java unit. Details of Phase ii will be shown in Section 3.

[2]https://www.eclipse.org/jdt/

Listing 1. Parse a Java source text string to find the AST of a Java unit

```java
public static ASTNode parserToAstFuncList(String sourceCodeFile, String
    funcName)
{
     ArrayList<ASTNode> AstFuncList = new ArrayList<>();
     ASTParser parser = ASTParser.newParser(AST.JLS8);
     parser.setSource(sourceCodeFile.toCharArray());
     parser.setKind(ASTParser.K_COMPILATION_UNIT);
     CompilationUnit cu = (CompilationUnit) parser.createAST(null);
     ASTVisitor visitor = new ASTVisitor()
     {
          @Override
          public boolean visit(TypeDeclaration node)
          {
               List<MethodDeclaration> methods = Arrays.asList(node.
                   getMethods());
               for (MethodDeclaration method : methods) {
                    if (method.isConstructor() == false)
                    {
                         AstFuncList.add(method);
                    }
               }

               return true;
          }
     };
     cu.accept(visitor);

     for (int i = 0; i < AstFuncList.size(); i++)
     {
          if (((MethodDeclaration)AstFuncList.get(i)).getName().
              getIdentifier().equals(funcName))
          {
               return AstFuncList.get(i);
          }
     }

     return null;
}
```

3. Generate the Corresponding Control Flow Graph for a Java Unit

Since this section presents many algorithms with several names representing CFG classes and utility functions, Tables 1 and 2 show these classes and functions being used, respectively. In addition, Figure 2 shows an overview of the CFG generation process. In this figure, the dash arrows from Algorithm Gen4ForStmt, Gen4ForEachStmt, Gen4WhileStmt, and Gen4DoStmt to Algorithm Gen4Block and

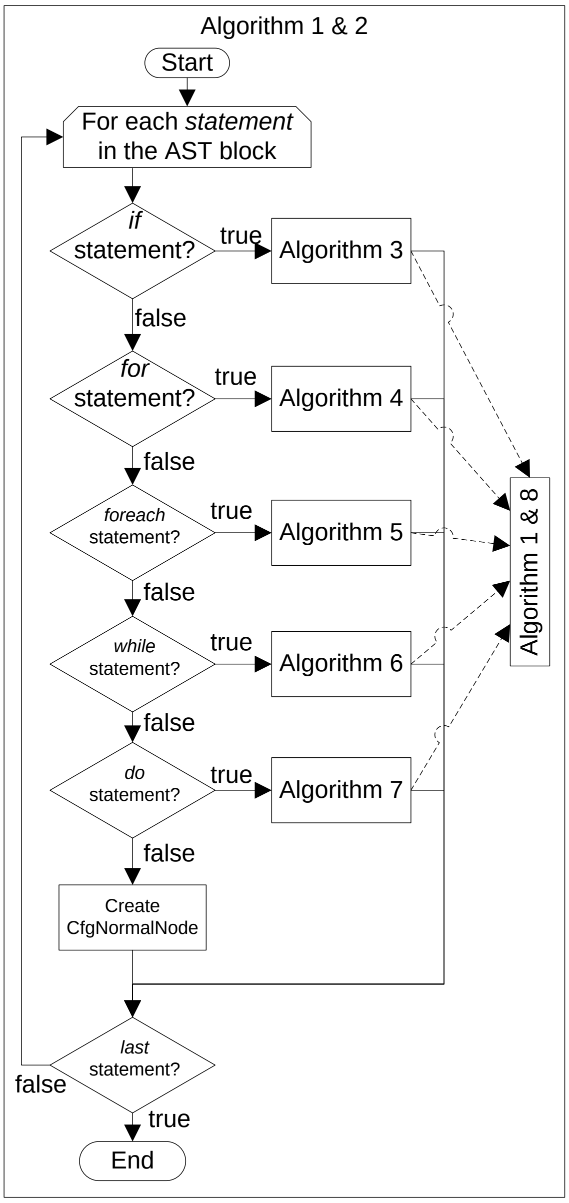


Figure 2. The overview of CFG generation process.

LinkSpecStmt show that there is a call from these algorithms to Algorithm Gen4Block and Gen4SpecStmt.

3.1. Generate CFG for A Java Unit

In Java, each block is a snippet of source code enclosed between two parentheses ‘{’ and ‘}’. For this reason, a unit function body is a block. There are many other code snippets, which are also called blocks. For example, for this *if* statement `if (a==‘c’){ return 0;} else { int x = 10;}`, there are two blocks corresponding to the cases where the condition `a==‘c’` is *true* or *false*.

After having the AST node associated with the given unit, we parse the node to get the AST corresponding to the body of the unit, which is an AST block. One block of AST can contain many statements. In turn, each statement can be a simple statement or contain many child statements. For this reason, the process of generating the CFG for a unit function is actually the process of generating the corresponding CFG for a block of statements.

The key idea of the process to generate CFG from an AST block is as follows. As we know, every statement or block always has a statement before and another statement after it. CFG4J generates the corresponding CFG of a given Java block, which is connected to its CFG nodes before and after it. Details of the process are shown in Algorithm 1 (denoted by Gen4Block).

The algorithm accepts an AST block (*astBlock*), the begin node (*beginNode*), the end node (*endNode*), the list of *continue* statement nodes (*contNodeList*), the list of *break* statement nodes (*breakNodeList*), and the list of *return* statement nodes (*retNodeList*) as its inputs.

At first, the algorithm gets the list of statement AST nodes which are direct children of *astBlock* and initializes the list of $contNodeList, breakNodeList, retNodeList$ with the empty set (∅) (lines 2-3). Then, for each statement (*statement*), the algorithm generates a CFG node for it by calling Algorithm Gen4Stmt (line 5). This job is responsible for generating CFG corresponding to each case of a simple statement, an *if* statement, a *for* statement, etc. The job returns the beginning node of the statement under consideration. In particular,

Table 1. CFG node class names

No.	Name	Description
1	CfgBeginForNode	A class representing a block of AST of a *for* statement
2	CfgBeginDoNode	A class representing a block of AST of a *do* statement
3	CfgBeginForEachNode	A class representing a block of AST of a *foreach* statement
4	CfgContNode	A class representing a block of AST of a *continue* statement
5	CfgBreakNode	A class representing a block of AST of a *break* statement
6	CfgRetNode	A class representing a block of AST of a *return* statement
7	CfgIfStmBlockNode	A class representing a block of AST of a *if* statement
8	getEndBlockNode	A class representing a CFG node that ends a block of CFG
9	CfgForStmBlockNode	A class representing a block of AST of a *for* statement
10	CfgForEachStmBlockNode	A class representing a block of AST of a *foreach* statement
11	CfgWhileStmBlockNode	A class representing a block of AST of a *while* statement
12	CfgDoStmBlockNode	A class representing a block of AST of a *do* statement
13	CfgBoolExprNode	A class representing a block of AST of a condition expression
14	CfgBlock	A class representing a block of AST
15	CfgNormalNode	A class representing a normal block of AST
16	CfgForEachExpressionNode	A class representing a block of AST of a *foreach* statement

Table 2. Utility function list

No.	Name	Description
1	get_statements	Get the list of statements from a given AST block
2	*create_a_new_CFG_node*	Create a new normal CFG node
3	getExpression	Get the expression of an *if*, *for*, *do*, or *while* AST block
4	getThenStatement	Get *then* expression of an *if* AST block
5	getElseStatement	Get *else* expression of an *if* AST block
6	setTrueNode	Set CFG node to the *true* case of a condition CFG node
7	setFalseNode	Set CFG node to the *false* case of a condition CFG node
8	initializers	Get the list of *initializers* of a given AST block
9	updaters	Get the list of *updater* of a given AST block
10	getAfterNode	Get the node right after a given CFG node
11	getParameter	Get the parameters of a given AST block
12	setHasMoreNode	Set the CFG node following a given node in case there is more node to process
13	setNoMoreNode	Set the CFG node following a given node in case there is no more node to process
14	getBody	Get the body block of a given AST block
15	Link	Create a directed edge from one CFG node to another

Algorithm 1: Generate CFG for an AST block

```
Input: astBlock: the input AST block
beginNode: CFG node before the block
endNode: CFG node after the block
contNodeList: continue node list
breakNodeList: break node list
retNodeList: return node list
Output: beginBlockNode: beginning node of the block
contNodeList: continue node list
breakNodeList: break node list
retNodeList: return node list
1  begin
2      statements ← get_statements(astBlock)
3      contNodeList, breakNodeList, retNodeList ← ∅
4      foreach statement in statements do
5          currentNode ← call Gen4Stmt( statement, beginNode, endNode);
6          if statement is the first statement then
7              beginBlockNode ← currentNode
8          end
9          if currentNode is CfgBoolExprNode or currentNode is CfgBeginForNode or
             currentNode is CfgBeginDoNode or currentNode is CfgBeginForEachNode then
10             beginNode ← currentNode.getEndBlockNode();
11         else
12             beginNode ← currentNode;
13         end
14         if currentNode is CfgContNode then
15             Link(currentNode, beginNode)
16             contNodeList ← contNodeList ∪ {currentNode}
17         else if (currentNode is CfgBreakNode) or (currentNode is CfgRetNode) then
18             Link(currentNode, endNode)
19             breakNodeList ← breakNodeList ∪ {currentNode}
20         end
21     end
22 end
```

it returns a *CfgBoolExprNode* node when processing an *if* statement, a *CfgBeginForNode* node when processing a *for* statement, etc. There are four types of nodes corresponding to compound statements as follows: (i) CfgBoolExprNode is the beginning node of *if* and *while* statements; (ii) CfgBeginForNode is the beginning node of *for* statement; (iii) CfgBeginDoNode is the beginning node of *do* statement; and (iv) CfgBeginForEachNode is the beginning node of *enhanced for* (for each) statement. This is also responsible for connecting the generated CFG with *beginNode* and *endNode*.

Later, the algorithm reassigns the *beginNode* to the end of the current node to prepare to generate CFG for the next statement (line 9-13). The algorithm also checks if the current node (*currentNode*) is either CfgContNode (a node corresponding to a *continue* statement), the algorithm links the current node with the beginning node (*beginNode*). This is the case of loop statements such as *for*, *for each*, *while*, and *do* statements. Those loop statements will be run from the beginning. Similar to the *continue* statement, if the current node is CfgBreakNode (a node corresponding to a *break* statement) or CfgRetNode (a node corresponding to a *return* statement), the algorithm links the current node to the end node (*endNode*). In this algorithm, *getEndBlockNode* method returns the end node of the corresponding generated CFG of a compound statement like *if* and *for* statements.

3.2. Generate CFG for One Statement

Given an AST node and the CFG nodes before (*beforeNode*) and after (*afterNode*) the current node, CFG4J generates the corresponding CFG (*currentNode*) for it. Then, CFG4J connects the generated CFG to *beforeNode* and *afterNode*. Basically, this is an interim step where the CFG creation process is redirected to either Algorithm Gen4IfStmt, Gen4ForStmt, Gen4ForEachStmt, Gen4WhileStmt, Gen4DoStmt for generating

CFG for *if*, *for*, *for each*, *while*, *do* statements or to create a normal CFG node for other simple statements. Details of the process are shown in Algorithm 2 (denoted by Gen4Stmt). In this algorithm, *CfgIfStmBlockNode*, *CfgForStmBlockNode*, *CfgForEachStmBlockNode*, *CfgWhileStmBlockNode*, *CfgDoStmBlockNode*, and *CfgNormalNode* are classes representing blocks of AST of *if*, *for*, *foreach*, *while*, *do*, and other kinds of statements, respectively.

The algorithm checks the given statement (*stm*) to see if it is in one of the following cases using *switch* statement. In case *stm* is an *IfStatement* AST node, the algorithm creates an instance *currentNode* of *CfgIfStmBlockNode* class by using the *new* keyword. Then, the algorithm calls Algorithm Gen4IfStmt to generate the CFG corresponding to *stm*. After that, the algorithm returns the first node (*beginIfNode*) of the generated CFG (lines 3-6).

In case *stm* is a *ForStatement* AST node, the algorithm creates an instance *currentNode* of *CfgForStmBlockNode* class. Then, the algorithm calls Algorithm Gen4ForStmt to generate the CFG corresponding to the *for* statement *stm*. After that, the algorithm returns the first node (*beginForNode*) of the generated CFG (lines 7-10).

In case *stm* is a *EnhancedForStatement* AST node, the algorithm creates an instance *currentNode* of *CfgForEachStmBlockNode* class. Then, the algorithm calls Algorithm Gen4ForEachStmt to generate the CFG corresponding to the *for each* statement *stm*. After that, the algorithm returns the first node (*beginForEachNode*) of the generated CFG (lines 11-14).

In case *stm* is a *WhileStatement* AST node, the algorithm creates an instance *currentNode* of *CfgWhileStmBlockNode* class. Then, the algorithm calls Algorithm Gen4WhileStmt to generate the CFG corresponding to the *for each*

Algorithm 2: Generate the corresponding CFG for one statement

```
Input: stm: the AST node under consideration;
beforeNode, afterNode: CFG nodes before &
after the block; contNodeList, breakNodeList,
retNodeList: continue, break, return node lists
Output: First node CFG if stm is a complex
        statement
or CFG node if stm is a simple statement;
contNodeList, breakNodeList, retNodeList:
continue, break, return node lists
1  begin
2      switch stm do
3          case IfStatement
4              currentNode ← new
                 CfgIfStmBlockNode(stm)
5              beginIfNode ← call Algorithm
                 Gen4IfStmt(currentNode,
                 beforeNode, afterNode,
                 contNodeList, breakNodeList,
                 returnNodeList);
6              return beginIfNode
7          case ForStatement
8              currentNode ← new
                 CfgForStmBlockNode(stm)
9              beginForNode ← call Algorithm
                 Gen4ForStmt(currentNode,
                 beforeNode, afterNode,
                 returnNodeList);
10             return beginForNode
11         case EnhancedForStatement
12             currentNode ← new
                 CfgForEachStmBlockNode(stm)
13             beginForEachNode ← call
                 Algorithm
                 Gen4ForEachStmt(currentNode,
                 beforeNode, afterNode,
                 returnNodeList);
14             return beginForEachNode
15         case WhileStatement
16             currentNode ← new
                 CfgWhileStmBlockNode(stm)
17             beginWhileNode ← call Algorithm
                 Gen4WhileStmt(currentNode,
                 beforeNode, afterNode,
                 returnNodeList);
18             return beginWhileNode
19         case DoStatement
20             currentNode ← new
                 CfgDoStmBlockNode(stm)
21             beginDoNode ← call Algorithm
                 Gen4DoStmt(currentNode,
                 beforeNode, afterNode,
                 returnNodeList);
22             return beginDoNode
23         otherwise do
24             currentNode ← new
                 CfgNormalNode(stm);
25         end
26     end
27     return currentNode
28 end
```

statement *stm*. After that, the algorithm returns the first node (*beginWhileNode*) of the generated CFG (lines 15-18).

In case *stm* is a *DoStatement* AST node, the algorithm creates an instance *currentNode* of *CfgDoStmBlockNode* class. Then, the algorithm calls Algorithm Gen4DoStmt to generate the CFG corresponding to the *for each* statement *stm*. After that, the algorithm returns the first node (*beginDoNode*) of the generated CFG (lines 19-22).

In case *stm* is other types of statements, the algorithm creates a new CFG node corresponding to the AST statement (*stm*) (line 24) and returns this instance (line 27).

3.3. Generate CFG for `if` Statement

Given an AST node corresponding to an *if* statement, CFG4J generates the CFG for it. Details of the process are shown in Algorithm 3 (denoted by Gen4IfStmt). The algorithm is also responsible for connecting the generated CFG with the nodes before and after it. In this algorithm, *CfgBoolExprNode* is a class representing a boolean expression that has two possible values of *true* and *false*. This class has two methods *setTrueNode* and *setFalseNode* to set the two nodes (true node and false node) corresponding to *true* and *false* cases.

The algorithm starts by retrieving the AST corresponding to the condition expression (*condAST*) and creates a CFG node for it (*condNode*) (line 2-3). Then, the algorithm links *condNode* with *beforeNode* (line 4). After that, the algorithm gets the AST node of *then* block, generates the corresponding CFG for it by calling Algorithm Gen4Block, and assigns its first node (*thenNode*) to the true node of *condNode* (lines 6). Later, the algorithm checks if the AST node of *else* block exists. If yes, it generates the corresponding CFG for it by calling Algorithm Gen4Block and assigns its first node (*elseNode*) to the false node of *condNode*

Algorithm 3: Generate the corresponding CFG for *if* statement

```
Input: ifAst: an AST node representing
       an if statement
beforeNode: CFG node before the statement
afterNode: CFG node after the statement
contNodeList: continue node list
breakNodeList: break node list
retNodeList: return node list
Output:
The first node of the generated CFG
contNodeList: continue node list
breakNodeList: break node list
retNodeList: return node list
1  begin
2      condAST ← ifAst.getExpression()
3      condNode ← new CfgBoolExprNode(condAST)
4      Link (condNode, beforeNode)
5      thenAST ← ifAst.getThenStatement()
6      thenNode ← call Algorithm Gen4Block(thenAST, condNode,
         afterNode, contNodeList, breakNodeList, returnNodeList)
       condNode.setTrueNode(thenNode)
7      if elseAST is not null then
8          elseAST ← ifAst.getElseStatement()
9          elseNode ← call Algorithm Gen4Block(elseAST,
             condNode, afterNode, contNodeList, breakNodeList,
             returnNodeList)
10         condNode.setFalseNode(elseNode)
11     else
12         condNode.setFalseNode(afterNode)
13     end
14     return condNode
15 end
```

(lines 7-10). If the AST node of *else* block does not exist, the algorithm assigns $afterNode$ into the false node of *condNode* (line 12). Finally, the algorithm returns *condNode* as the first node of the CFG corresponding to *if* statement under consideration (line 14).

3.4. Generate CFG for `for` *Statement*

In Java language, *for* statement is used to loop a certain block of statements through a list of items using one counting variable. Particularly, *for* statement refers to this kind of statement: `for (i = 0; i < 10; i++){//Do something}`. In this *for* statement, there are four parts: initializers (`i = 0`), condition expression (`i < 10`), updaters (`i++`), and body (`//Do something`). In which, initializers and updaters are lists of sub-statements. The process of parsing the *for* AST node into the corresponding CFG follows the Java *for* statement execution order. Details of the process are shown in Algorithm 4 (denoted by Gen4ForStmt).

Algorithm Gen4ForStmt accepts the AST node ($forAst$) corresponding to the *for* statement under consideration, the CFG nodes before and after the *for* statement, and the *return* node list as its inputs. The algorithm starts by processing the list of initializers (line 3-7). Each initializer will be parsed as one CFG node. Then, they are connected to each other. The algorithm uses a temporary node *tempBeforeNode* (line 2) to connect one current initializer node to its before node. This is also used to connect the first initializer node to *beforeNode*.

Later, the condition node is created and linked to the last node of the initializers (lines 8-10). After that, the body block node and updater nodes are created (lines 11-22). Like the initializer nodes, updater nodes are created and linked to each others using a temporary node (named *tempNode*) and another temporary node (*firstUpdaterNode*) for keeping track of the first updater node. The algorithm links the end

Algorithm 4: Generate the corresponding CFG for *for* statement

```
Input: $forAst$: the for AST node
$beforeNode$: CFG node before the block
$afterNode$: CFG node after the block
$retNodeList$: return node list
Output: the first node of $for$ CFG
1  begin
2      $tempBeforeNode \leftarrow beforeNode$
3      foreach init in $forAst$.initializers() do
4          $normalNode \leftarrow$ new CfgNormalNode(init)
5          Link($tempBeforeNode$, $normalNode$, $afterNode$)
6          $tempBeforeNode \leftarrow normalNode$
7      end
8      $condAST \leftarrow forAst$.getExpression()
9      $condNode \leftarrow$ new CfgBoolExprNode($condAST$)
10     LinkCurrentNode($tempBeforeNode$, $condNode$, $afterNode$)
11     $bodyAst \leftarrow forAst$.getBody()
12     $bodyBlockNode \leftarrow$ new CfgBlock($bodyAst$)
13     $tempNode \leftarrow bodyBlockNode$
14     $firstUpdaterNode \leftarrow null$
15     foreach upd in $forAst$.updaters() do
16         $normalNode \leftarrow$ new CfgNormalNode(upd)
17         Link($tempNode$, $normalNode$, $afterNode$)
18         $tempNode \leftarrow normalNode$
19         if upd is the first updater then
20             $firstUpdaterNode \leftarrow normalNode$
21         end
22     end
23     Link ($tempNode$, $condNode$)
24     $bodyNode \leftarrow$ call Algorithm Gen4Block($bodyAst$, $condNode$, $firstUpdaterNode$, $contNodeList$, $breakNodeList$, $returnNodeList$);
25     call Algorithm LinkSpecStmt($contNodeList$, $breakNodeList$, $returnNodeList$)
       $condNode$.setTrueNode($bodyNode$)
26     $condNode$.setFalseNode($afterNode$)
27     return $beforeNode$.getAfterNode()
28 end
```

updater node (*tempNode*) with the condition node (*condNode*).

The body CFG is built from the body block node by calling Algorithm Gen4Block (line 24). Algorithm Gen4Block is also responsible for linking the first node and end node of the body CFG to the condition node and the first updater node. Then, the algorithm links all nodes corresponding to *continue*, *break*, and *return* statements to either *beforeNode* or *afterNode* of the *for* statement by calling Algorithm LinkSpecStmt (line 25). Finally, the first and last nodes of the body CFG (*afterNode*) are set to true and false nodes of the condition node, respectively (lines 25-26). The algorithm returns the first node of the CFG corresponding to *for* statement (i.e., the node after *beforeNode*) (line 27).

3.5. Generate CFG for `foreach` Statement

Let us continue building CFG for the *foreach* statement. In Java, *foreach* statement is also used to loop a certain process through a list of items using one running variable. Particularly, *foreach* statement is the following kind of statement: `for (Type str: list){//Do something}`. In which, *str* (called parameter) is a running variable that gets each value from the list (*list*) for every loop. Next, *list* (called expression) is an expression that returns a list of values. Last, *Do something* (called body) is a block of statements to execute for each value in the list. CFG4J builds CFG for *foreach* statement according to its execution order. Details of the process are presented in Algorithm 5 (denoted by Gen4ForEachStmt).

Algorithm 5: Generate the corresponding CFG for *foreach* statement

```
Input: feAst: The AST node under consideration
beforeNode: CFG node right before the block
afterNode: CFG node right after the block
retNodeList: return node list
Output: The first node of foreach CFG
1  begin
2      exprAST ← feAst.getExpression()
3      paramAST ← feAst.getParameter()
4      exprNode ← new CfgForEachExpressionNode(exprAST, paramAST)
5      Link (beforeNode, exprNode)
6      bodyBlock ← feAst.getBody();
7      bodyBlockNode ← new CfgBlock(bodyBlock);
8      bodyNode ← call Algorithm Gen4Block(bodyBlockNode, exprNode, exprNode, contNodeList, breakNodeList, returnNodeList)
9      call Algorithm LinkSpecStmt(contNodeList, breakNodeList, returnNodeList)
10     exprNode.setHasMoreNode(bodyNode)
11     exprNode.setNoMoreNode(afterNode)
12     return exprNode
13 end
```

The algorithm accepts an *EnhancedForStatement* AST node, the CFG nodes before and after the current node, and the *return* node list as its inputs. The algorithm starts by getting the AST of the expression and parameter of *feAst* to create the corresponding expression node and link it to *beforeNode* (lines 2-5). Then, the algorithm gets the body block of statements, creates the corresponding body CFG, and links it with *exprNode* (lines 6-8). In this step, the first and last nodes of the body CFG are also linked to the expression node (*exprNode*) according to how *foreach* statement works. Then, the algorithm links all nodes corresponding to *continue*, *break*, and *return* statements to either *beforeNode* or *afterNode* of the *for* statement (line 9). In this algorithm, *exprNode* has two special nodes to specify its successor nodes in cases where there is more node(s) or no more node, respectively. The algorithm sets the first node of the body CFG (*bodyNode*) and *afterNode* to the two successor nodes of *exprNode*, respectively (lines 10-11). Finally, the algorithm returns *exprNode* as the first node of the *foreach* CFG (line 12).

3.6. Generate CFG for `while` Statement

In Java, *while* statement is used to loop a certain block of statements with an unknown number of iterations. Following is one example of a *while* statement: `while (i < 10){//Do something}`. In which, `i < 10` is called the condition expression and `//Do something` is called the body of the *while* statement. CFG4J also builds the CFG for the *while* statement by following its execution order. Details of the process are shown in Algorithm 6 (denoted by Gen4WhileStmt).

The algorithm accepts the corresponding AST block of the *while* statement, the CFG nodes before and after the current node, and the *return* node list as its inputs. At first, the algorithm gets the condition expression AST, creates a CFG node for it, and links it with the node before (*beforeNode*) (lines 2-4). After that, the AST corresponding to the body block is retrieved, the body CFG is built by calling Algorithm Gen4Block (lines 5-6). This step is responsible for connecting the newly built CFG to its before and after nodes (both nodes are *condNode*). Later, the algorithm links all nodes corresponding to *continue*, *break*, and *return* statements to either *beforeNode* or *afterNode* of the *for* statement (line 7). Then, the algorithm sets *bodyNode* and *afterNode* to true and false nodes of *condNode*, respectively (lines 8-9). Finally, the algorithm returns *condNode* (line 10.

Algorithm 6: Generate the corresponding CFG for *while* statement

```
Input: $whileAst$: the AST node under consideration
$beforeNode$: CFG node right before the block
$afterNode$: CFG node right after the block
$retNodeList$: return node list
Output: the first node of the while CFG
1  begin
2      $condAST \leftarrow whileAst$.getExpression()
3      $condNode \leftarrow$ new CfgBoolExprNode($condAST$)
4      Link ($beforeNode$, $condNode$)
5      $bodyBlock \leftarrow whileAst$.getBody()
6      $bodyNode \leftarrow$ call Algorithm Gen4Block($bodyBlock$, $condNode$, $condNode$, $contNodeList$, $breakNodeList$, $returnNodeList$)
7      call Algorithm LinkSpecStmt($contNodeList$, $breakNodeList$, $returnNodeList$)
8      $condNode.setTrueNode(bodyNode)$
9      $condNode.setFalseNode(afterNode)$

10     return $condNode$
11 end
```

3.7. Generate CFG for `do` Statement

Let us consider *do* statement. This is used to loop a certain block of statements until a predefined condition is no longer correct. Particularly, *do* statement refers to the following kind of statement: `do{//Do something} while (i < 10)`. In this statement, `//Do something` is the block of statements we want to loop through, and `i < 10` is the condition expression. If this condition expression is true, the block `//Do something` will be executed.

CFG4J constructs the corresponding CFG of *do* statement according to its execution order. The CFG construction process details are shown in Algorithm 7 (denoted by Gen4DoStmt).

Algorithm 7: Generate the corresponding CFG for *do* statement

```
Input: $doAst$: the AST node under consideration
$beforeNode$: CFG node right before the block
$afterNode$: CFG node right after the block
$retNodeList$: return node list
Output: the first node of do CFG
1  begin
2      $condAST \leftarrow doAst$.getExpression()
3      $condNode \leftarrow$ new CfgBoolExprNode($condAST$)
4      $bodyAst \leftarrow doAst$.getBody()
5      $bodyNode \leftarrow$ call Algorithm Gen4Block($bodyAst$, $beforeNode$, $condNode$, $contNodeList$, $breakNodeList$, $returnNodeList$)
6      call Algorithm LinkSpecStmt($contNodeList$, $breakNodeList$, $returnNodeList$)
7      $condNode.setTrueNode(bodyNode)$
8      $condNode.setFalseNode(afterNode)$
9      return $bodyNode$
10 end
```

The algorithm accepts the AST block of *do* statement, the CFG nodes before and after the current node as its inputs. First, the algorithm gets the condition expression AST and creates the CFG node corresponding to it (lines 2-3). Then, the algorithm gets the body AST, creates the corresponding CFG for it by calling Algorithm Gen4Block. This step is responsible for creating the connection between the newly created CFG with $beforeNode$ and $condNode$, respectively (lines 4-5). Then, the algorithm links all nodes corresponding to *continue*, *break*, and *return* statements to either *beforeNode* or *afterNode* of the *for* statement (line 6). Later, the algorithm sets $bodyNode$ and $afterNode$ to true and false nodes of $condNode$, respectively (lines 7-8). Finally, the algorithm returns $bodyNode$ as the first node of the generated CFG corresponding to *do* statement (line 9).

3.8. Link continue, break, and return Statements

In Java projects, there are three special statements that affect the generated CFG: *continue*, *break*, and *return*. When reaching these statements, the control flow needs to break out of the current loop, continue another loop from the beginning, or go to the end of the unit under test. Algorithm LinkSpecStmt shows details of the linking process between these statements to either the beginning node or the ending node of a certain block. The process of generating CFG for loop statements such as *for*, *foreach*, *while*, *do*, will be responsible for parsing the correct *beforeNode* and *afterNode* to Algorithm 8 (denoted by LinkSpecStmt).

Algorithm LinkSpecStmt accepts the node right before the block (*beforeNode*), the node right after the block (*afterNode*), a list of *continue* statement nodes (*contNodeList*), a list of *break* statement nodes (*breakNodeList*), and a list of *return* statement nodes *retNodeList* as its inputs. At the start, the algorithm connects all nodes corresponding to *continue* statements to the *beforeNode* (lines 2-4). For those nodes corresponding to *break* statements, the algorithm connects them to the end node of the block (*afterNode*) (lines 5-7). Finally, the algorithm connects all nodes corresponding to *return* statements to the end node of the block (lines 8-10).

Algorithm 8: Link *continue*, *break*, and *return* statements

```
Input: $beforeNode$: CFG node right before the block
       $afterNode$: CFG node right after the block
       $contNodeList$: continue node list
       $breakNodeList$: break node list
       $retNodeList$: return node list
1  begin
2      foreach $contNode$ in $contNodeList$ do
3          Link ($contNode$, $beforeNode$)
4      end
5      foreach $breakNode$ in $breakNodeList$ do
6          Link ($breakNode$, $afterNode$)
7      end
8      foreach $retNode$ in $retNodeList$ do
9          Link ($retNode$, $afterNode$)
10     end
11 end
```

3.9. Complexity

As we can see, the process of generating the CFG from an AST block corresponding to a Java unit is a recursive process (i.e., calling the Algorithm Gen4Block recursively). This process depends on the number of direct children statements of the block. Let m be the maximum number of direct children statements of all blocks being processed and n be the number of recursive calls. The time complexity of the whole CFG generation process is $O(m * n)$.

3.10. An Example

Let's consider an example of the CFG generation process for `getAverage` function shown in Listing 2. The CFG generation process is shown in Table 3. In this table, for a short description of a block (i.e., the source code from a curly brace opening character '{' to the corresponding line with the curly brace closing character '}'), we describe it from the line with '{' to the line with '}'. In addition, we named the statements in a certain block from S_1 to S_n, where n is the number of statements in that block. For reading convenience, a condition expression, an initializer of a for statement, and an updater of a for statement are named with 'C', 'I', and 'U', respectively. Results are shown in Figure 3 and 4.

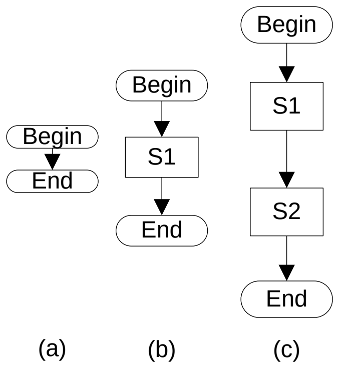


Figure 3. The CFG being created.

Listing 2. The getAverage function

```
float getAverage(int arr[],int n){
    float avg = 0;
    int temp = 0;
    if (n > 0){
        for(int i=0; i<n; i++){
            temp += arr[i];
        }
        avg = temp / n;
    }
    return avg;
}
```

4. Experiments

4.1. Experimental results

To evaluate CFG4J, we have implemented the method in Java language in a tool named

Table 3. An example of the CFG generation process

Running algorithm	**Processing statement**	**Result**

Before start	beginNode = Begin,
	endNode = End,
	astBlock = B1,
	other lists are ∅
		The current CFG is shown in Figure 3.a

1st call of Algorithm Gen4Block	B1 = Block line 1 to 11	Call Algorithm Gen4Stmt
Algorithm Gen4Stmt	S1 = “float avg = 0;”	+ CFG node S1 of CfgNormalNode is created + The current CFG is shown in Figure 3.b
Algorithm Gen4Stmt	S2 = “int temp = 0;”	+ CFG node S2 of CfgNormalNode is created + The current CFG is shown in Figure 3.c
Algorithm Gen4Stmt	S3 = if statement from line 4 to 9	Call Algorithm Gen4IfStmt
Algorithm Gen4IfStmt	if statement from line 4 to 9	+ CFG node S3.C of CfgBoolExprNode is created + 2nd call of Algorithm Gen4Block + The current CFG is shown in Figure 4.a
2nd call of Algorithm Gen4Block	S3.T = Block line 4 to 9	Call Algorithm Gen4ForStmt
Algorithm Gen4ForStmt	*S3.T.S1 = for statement from line 5 to 7*	+ CFG nodes S3.T.S1.I of CfgNormalNode is created + CFG node S3.T.S1.C of CfgBoolExprNode is created + CFG node S3.T.S1.U of CfgNormalNode is created + 3rd call of Algorithm Gen4Block + The current CFG is shown in Figure 4.b
3rd call of Algorithm Gen4Block	S3.T.S1.B = Block line 5 to 7	Call Algorithm Gen4Stmt
Algorithm Gen4Stmt	S3.T.S1.B.S1 = “temp += arr[i];”	+ CFG node S3.T.S1.B.S1 of CfgNormalNode is created + Return to 3rd call of Algorithm Gen4Block + The current CFG is shown in Figure 4.c
3rd call of Algorithm Gen4Block		+ Return to 2rd call of Algorithm Gen4Block
2rd call of Algorithm Gen4Block	S3.T.S2 = “avg = temp / n;”	+ CFG node S3.T.S2 of CfgNormalNode is created + Return to 1rd call of Algorithm Gen4Block + The current CFG is shown in Figure 4.d
1st call of Algorithm Gen4Block	S4 = “return avg ;”	+ CFG node S4 of CfgNormalNode is created + The current CFG is shown in Figure 4.e + Return and stop

CFG4J Tool and done some assessments about its applicability and effectiveness. The source code of the tool is published at `https://github.com/vnuvietth/CFG4J_paper`. We focus on the key factors of the CFG building process, which are the required time and memory

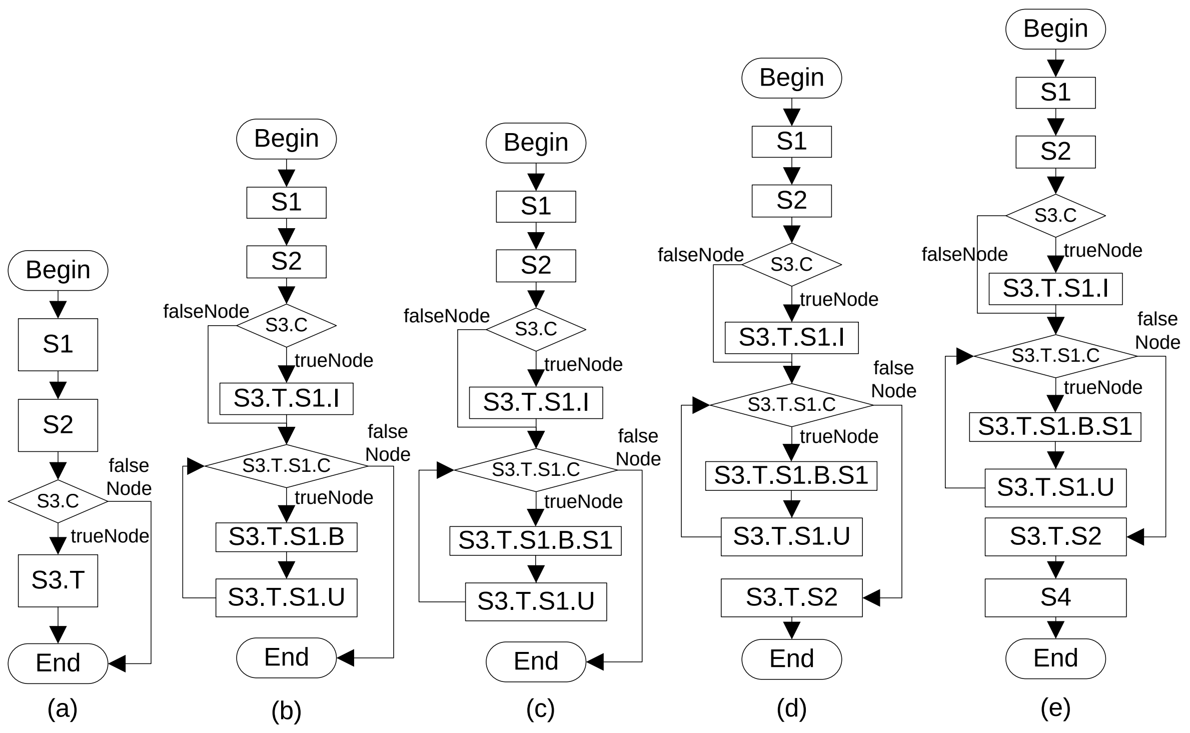


Figure 4. The CFG being created (cont.).

usage. The experiments are performed on a machine that has the following configuration: Processor: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz, 2201 Mhz, 2 Core(s), 4 Logical Processor(s); RAM: 8GB; Operating system: Microsoft Windows 10 Home. We have used some units from a repository named The Algorithms - Java[3] and the website `https://www.geeksforgeeks.org` for testing. The test units are as follows:

- LinearSearch: Retrieved from The Algorithms - Java. Given an array of n elements, the function is to search a given element x in the array using linear search.
- LeapYear: Retrieved from `https://www.geeksforgeeks.org`. The function is to check if a year. As we know, a year is a leap year if the following conditions are satisfied: (i) The year is a multiple of 400, and (ii) The year is a multiple of 4 and not a multiple of 100.
- FibonacciSearch: Retrieved from The Algorithms - Java. Given an array of n elements, the function is to search a given element x in the array using a Fibonacci search.
- JumpSearch: Retrieved from The Algorithms - Java. Given an array of n elements, the function is to search a given element x in the array using a jump search.
- SelectionSort: Retrieved from The Algorithms - Java. Given an array of n elements, the function is to sort values in the array using the selection sort.
- BubbleSort: Retrieved from The Algorithms

[3]https://github.com/TheAlgorithms/Java

- Java. Given an array of n elements, the function is to sort values in the array using bubble sort.

- SimpleSort: Retrieved from The Algorithms - Java. Given an array of n elements, the function is to sort values in the array using the simple sort.
- Average: Retrieved from The Algorithms - Java. Given an array of n elements, the function is to calculate the average of all values in the array.
- BinaryPower: Retrieved from The Algorithms - Java. Given two numbers a and p, the function is to calculate the power a^p using the binary operator.

To remove the overhead dependencies when performing the experiments, we have done experiments for each testing function ten times and recorded the result. Experimental results are shown in Table 4. In Table 4, the columns are as follows.

- “No.”: the ordinal number of the testing function in our experiments.
- “Function name”: the function name of the testing function in our experiments.
- “Time (ms)”: the required time (in milliseconds) to generate CFG for the corresponding function.
- “Memory (MB)”: the required memory (in Megabytes) to generate CFg for the corresponding function.

Table 4. Experimental results

No.	**Function name**	**Time (ms)**	**Memory (MB)**
1	LinearSearch	38.0	8.51
2	LeapYear	26.0	9.11
3	FibonacciSearch	31.0	8.94
4	JumpSearch	33.0	8.06
5	SelectionSort	21.0	7.18
6	BubbleSort	40.0	7.69
7	SimpleSort	22.0	8.85
8	Average	34.0	9.46
9	BinaryPower	29.0	9.09

4.2. Discussions

From the experimental results shown in Table 4, we have the following observations.

- The CFG generation for all functions requires less than 100ms. This is an acceptable amount of time as a pre-processing time in addition to the time required by other kinds of methods such as test data generation, code coverage analysis, code highlight, etc. which accepts CFG as an input. The required amount of time when generating CFG for testing functions is not much different from each other.
- The memory usage when generating CFG for those testing functions is quite small and not much different from each other. This implies the memory usage efficiency of CFG4J when generating CFG. This is a good start when implementing other analysis methods, which use CFG.
- The memory and time usages are not the same for all functions under test as shown in Table 4 as those results depend on the overhead and computer status when testing.

From the time and memory usage, we can see that CFG4J is applicable in combination with other analysis methods or test data generation for real-world projects.

4.3. Threats to Validity

As described in the above sections, there are some threats that can affect the validity of

CFG4J. First, CFG4J can only work with units whose statements are executed sequentially. This does not affect the applicability of CFG4J as we are building CFG for Java units. Those units are supposed to be called in other components of a project. For this reason, each unit can be considered as one complex statement and is executed once as a whole. However, currently, CFG4J has not supported those statements which invoke the concurrent process in Java. To do this, first, we need a formal definition of how concurrent processes can be presented using CFG. Then, we can refine CFG4J to make it support those statements. However, this is not in the scope of this paper.

Second, CFG4J depends on the JDT libraries to parse a given source code to generate the required AST. However, JDT libraries support Java version 5 to the latest version. As a result, almost all Java projects are supported by CFG4J except those statements mentioned in the first and fourth points). This will not be an obstacle to preventing CFG4J from being applied to projects in practice.

Third, regarding the soundness of CFG4J, we see that Java is an object-oriented programming language supporting structural programming. That is Java supports the three main types of statements sequence, condition, and loop. From a given AST, we can retrieve all the necessary information. For this reason, CFG4J is sound. The given AST may not contain all information for other purposes. However, for the generation of CFG, AST has all the necessary information.

Forth, in this version of the method, we have presented the process for generating CFG corresponding to common statements like simple statements (i.e., declaration statements, assignment statements, etc.), branch statements (i.e., `if` statements), and loop statements (i.e., `for`, `foreach`, `while`, `do/while` statements). There are many special statements in Java that have not been addressed in this paper such as `switch`, lambda expression, anonymous class, object method call, etc. However, we think that the method can be extended to generate CFG for these statements. The reason is that CFG is actually the visualization of the statement execution flow.

Last but not least, the version of Java source code under testing can affect the validity of the CFG4J method. If the JDT libraries cannot parse the given Java source code to generate the required AST, the CFG4J cannot work. However, once the JDT libraries can generate the required AST, CFG4J can generate the required CFG.

5. Related Works

In 2018, Zambon and Rensink proposed a method named recipes for compositional construction of CFG for Java 6 [22]. The idea of this method is based on the concept of recipes, which are essentially procedures with atomic behavior. In turn, those recipes can recursively compose new recipes. Although the idea of recipes is good for constructing CFG for Java, there are still some limitations. First, it is based on the self-constructed component called *graph compiler* to produce the AST for the Java program under analysis. This limits the tool to be applied only to Java 6. To apply to a higher version of Java, this component must be upgraded. Second, although the program ASTs generated by the graph compiler conform to the Java AST type-graph, the key idea of recipes when constructing CFG is to add flow edges between AST nodes.

Sharing the same interest as Zambon and Rensik about building CFG for Java programming language, we focus on building CFG for units only. There are four main differences between CFG4J and recipes as follows. First, CFG4J focuses on building CFG for Java units, not for the whole Java program as recipes. The generated CFG will be used as inputs for other analysis methods such as test data generation, code coverage calculation, etc. Second, CFG4J is based on the recursive idea when parsing Java code blocks, which are code snippets surrounded by the two parentheses of '{' and '}', as the main elements of the method. Third, CFG4J makes use of parsers of the Eclipse Java development tools (JDT) to parse the given Java unit into AST blocks. By using these parsers, we have the latest supported Java version whenever these parsers support. We argue that we should reuse the available parsers to save effort and have the best support rather than rebuilding a new parser to support all versions of Java. This is especially true when a new Java version emerges, all parsers need to be upgraded. Finally, CFG4J builds CFG for units based on Java blocks of AST corresponding to the unit source code by parsing through them instead of adding flow edges between AST nodes as in recipes.

In 2004, Jang-Wu and Byeong-Mo proposed a method to construct CFG by computing the normal and exception flows separately. Their paper has two contributions. First, they show that normal flow and exception flow can be safely decoupled and hence computed separately by examining fourteen Java programs. Second, they proposed an analysis that estimates the exception control flow and an exception flow graph that represents the exception control flows. They show that a control flow graph can be constructed by merging an exception flow graph onto a normal flow graph [23].However, the paper does not show the details of how to construct the control flow graph from a given Java source code. We argue that the method does not provide a general method for constructing CFG for Java programs. Sharing the same interest as Jang-Wu and Byeong-Mo in constructing CFG for Java programs, we focus on providing a detailed method to construct CFG for Java units. CFG4J builds CFGs for units by using a recursive method and for Java's three main types of control statements, which are sequence, branch, and loop statements.

In 2016, Afshin et al. proposed an algorithm to extract flow graphs from Java bytecode, including exceptional control flows [24, 25]. This method is fundamentally different from CFG4J that this method is based on Java bytecode analysis while CFG4J uses Java source code as inputs. In addition, CFG4J focuses on generating CFG for units using the corresponding AST of these units. CFG4J employs a recursive method to parse Java code blocks for building the required CFG.

In 2006, Zaretsky et al. proposed a method to generate the control and data flow graph from the schedule and pipelined assembly code [26]. The method consists of three stages: generating a control flow graph, linearizing the assembly code, and generating the data flow graph. Sharing the same interest as Zaretsky to generate CFG for Java source code, CFG4J is different from Zaretsky's method in that Zaretsky's method uses the assembly code while CFG4J makes use of Java unit source code to generate CFG.

In 2014, Pedro et al. proposed a method for extracting CFGs incrementally from incomplete Java byte code [27]. These CFGs are provably sound with regard to sequences of method invocations and exceptions. These generated CFGs are suitable for many program analysis methods such as model checking of temporal control flow safety properties. The soundness of these CFGs comes at the price of over-approximation. Sometimes, it gives false positive reports during verification. In addition, the method supports incremental refinements of the extracted models when the components are evolved. Sharing the same interest as Zaretsky to generate CFG for Java programs, CFG4J is based on the source code directly and aims to CFGs being used in other program analysis methods such as test data generation methods or model-based testing of the unit under check.

Table 5. Related works comparison

Paper	Key point	Limitations
CFG4J	From Java unit source code	Some statements are not supported yet
Zambon and Rensink [22]	From recipes for the Java program	- Self-constructed graph compiler to generate AST - To apply to newer version than Java 6, the compiler must be upgraded
Jang-Wu and Byeong-Mo [23]	Merging normal flow and exception flows of CFG	- Not provide details about the CFG construction process from source code
Afshin et al. [24, 25]	From Java byte code	The method is not based on Java source code
Zaretsky et al. [26]	From the schedule and pipelined assembly code	The method is not based on Java source code
Pedro et al. [27]	From incomplete Java byte code	- The method is not based on Java source code - Can contain false positive

6. Conclusion

We have presented a method named CFG4J for the CFG generation of Java units. The method uses JDT libraries to generate the corresponding AST of the given Java unit. Then, the AST is used in the CFG generation process for three main types of statements: sequence, condition, and loop. The initial experimental results show that the time and memory usage of CFG4J are acceptable when being used as inputs for other analysis methods.

Although CFG4J can generate CFG for the three main common types of statements in Java units, there is much work to do. First, we need to perform more experiments with bigger Java projects in practice to find out more special statements that have not been processed. Second, we are in the process of implementing other kinds of analysis methods using CFGs as inputs such as symbolic execution, test data generation, etc. Finally, we are also implementing a friendlier user interface for the tool so that other software engineers and researchers can have a reference to CFG4J. For this reason, we believe that CFG4J will contribute more value to both the software industry and researcher communities.

References

[1] D. A. Duffy, Principles of Automated Theorem Proving, John Wiley & Sons, Inc., New York, NY, USA, 1991.

[2] E. W. Dijkstra, Guarded Commands, Nondeterminacy and Formal Derivation of Programs, Commun. ACM 18 (8) (1975) 453–457.

[3] C. A. R. Hoare, An Axiomatic Basis for Computer Programming, Commun. ACM 12 (10) (1969) 576–580.

[4] D. Kapur, M. Subramaniam, Lemma Discovery in Automating Induction, in: M. A. McRobbie, J. K. Slaney (Eds.), Automated Deduction — Cade-13, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 538–552.

[5] M. Kaufmann, J. Moore, Some Key Research Problems in Automated Theorem Proving for Hardware and Software Verification 98 (2004) 181—-196.

[6] M. Sipser, Introduction to the Theory of Computation, 1st Edition, International Thomson Publishing, 1996.

[7] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, Systems and Software Verification: Model-Checking Techniques and Tools, 1st Edition, Springer Publishing Company, Incorporated, 2010.

[8] E. M. Clarke, Jr., O. Grumberg, D. A. Peled, Model Checking, MIT Press, Cambridge, MA, USA, 1999.

[9] L. Copeland, A Practitioner's Guide to Software Test Design, Artech House, Inc., USA, 2003.

[10] H.-V. Tran, P. N. Hung, V.-H. Nguyen, T. Aoki, A Framework for Assume-Guarantee Regression

Verification of Evolving Software, Science of Computer Programming 193 (2020) 102439. doi:https://doi.org/10.1016/j.scico.2020.102439.

[11] H. Tran, P. N. Hung, V. H. Nguyen, On Locally Minimum and Strongest Assumption Generation Method for Component-Based Software Verification, IEICE Trans. Inf. Syst. 102-D (8) (2019) 1449–1461. doi:10.1587/transinf.2018FOP0004.

[12] J. M. Cobleigh, D. Giannakopoulou, C. S. Păsăreanu, Learning Assumptions for Compositional Verification, in: Proceedings of the 9th Int. Conf. on Tools and Alg. for the Constr. and Anal. of Sys., TACAS'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 331–346.

[13] D. Giannakopoulou, C. S. P"s"reanu, H. Barringer, Assumption Generation for Software Component Verification, in: Proceedings of the 17th IEEE International Conference on Automated Software Engineering, ASE '02, IEEE Computer Society, USA, 2002, p. 3. doi:10.1109/ASE.2002.1114984.

[14] D.-A. Nguyen, P. N. Hung, A Test Data Generation Method for C/C++ Projects, in: Proceedings of the Eighth International Symposium on Information and Communication Technology, SoICT 2017, Association for Computing Machinery, New York, NY, USA, 2017, p. 431–438. doi:https://doi.org/10.1145/3155133.3155144.

[15] D.-A. Nguyen, T. N. Huong, H. V. Dinh, P. N. Hung, Improvements of Directed Automated Random Testing in Test Data Generation for C++ Projects, International Journal of Software Engineering and Knowledge Engineering 29 (09) (2019) 1279–1312. doi:10.1142/S0218194019500402.

[16] B. Botella, A. Gotlieb, C. Michel, Symbolic execution of floating-point computations: Research Articles, Softw. Test. Verif. Reliab. 16 (2) (2006) 97–121. doi:10.5555/1133626.1133628.

[17] S. Khurshid, Y. L. Suen, Generalizing Symbolic Execution to Library Classes, SIGSOFT Softw. Eng. Notes 31 (1) (2005) 103–110. doi:https://doi.org/10.1145/1108768.1108817.

[18] P. Godefroid, N. Klarlund, K. Sen, DART: Directed Automated Random Testing, PLDI '05, Association for Computing Machinery, New York, NY, USA, 2005, p. 213–223. doi:https://doi.org/10.1145/1064978.1065036.

[19] R. Gupta, M. L. Soffa, J. Howard, Hybrid Slicing: Integrating Dynamic Information with Static Analysis, ACM Trans. Softw. Eng. Methodol. 6 (4) (1997) 370–397. doi:https://doi.org/10.1145/261640.261644.

[20] H. Cleve, A. Zeller, Locating Causes of Program Failures, in: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, Association for Computing Machinery, New York, NY, USA, 2005, p. 342–351. doi:https://doi.org/10.1145/1062455.1062522.

[21] H. Nilsson, P. Fritzson, Lazy Algorithmic Debugging: Ideas for Practical Implementation, in: P. A. Fritzson (Ed.), Automated and Algorithmic Debugging, Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, pp. 117–134. doi:https://doi.org/10.1007/BFb0019405.

[22] E. Zambon, A. Rensink, Recipes for Coffee: Compositional Construction of JAVA Control Flow Graphs in GROOVE, Springer International Publishing, Cham, 2018, pp. 305–323. doi:10.1007/978-3-319-98047-8_19.

[23] J. Jo, B. Chang, Constructing Control Flow Graph for Java by Decoupling Exception Flow from Normal Flow, in: A. Laganà, M. L. Gavrilova, V. Kumar, Y. Mun, C. J. K. Tan, O. Gervasi (Eds.), Computational Science and Its Applications - ICCSA 2004, International Conference, Assisi, Italy, May 14-17, 2004, Proceedings, Part I, Vol. 3043 of Lecture Notes in Computer Science, Springer, 2004, pp. 106–113. doi:https://doi.org/10.1007/978-3-540-24707-4_14.

[24] A. Amighi, P. de Carvalho Gomes, D. Gurov, M. Huisman, Provably Correct Control Flow Graphs from Java Bytecode Programs with Exceptions, Int. J. Softw. Tools Technol. Transf. 18 (6) (2016) 653–684. doi:https://doi.org/10.1007/s10009-015-0375-0.

[25] A. Amighi, P. de C. Gomes, D. Gurov, M. Huisman, Sound Control-Flow Graph Extraction for Java Programs with Exceptions, in: G. Eleftherakis, M. Hinchey, M. Holcombe (Eds.), Software Engineering and Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 33–47. doi:https://doi.org/10.1007/978-3-642-33826-7_3.

[26] D. C. Zaretsky, G. Mittal, R. Dick, P. Banerjee, Generation of Control and Data Flow Graphs from Scheduled and Pipelined Assembly Code, in: E. Ayguadé, G. Baumgartner, J. Ramanujam, P. Sadayappan (Eds.), Languages and Compilers for Parallel Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 76–90. doi:https://doi.org/10.1007/978-3-540-69330-7_6.

[27] P. de Carvalho Gomes, A. Picoco, D. Gurov, Sound Control Flow Graph Extraction from Incomplete Java Bytecode Programs, in: S. Gnesi, A. Rensink (Eds.), Fund. Approaches to Soft. Eng., Springer Berlin Heidelberg, 2014, pp. 215–229. doi:10.1007/978-3-642-54804-8_15.