Proceedings of the World Congress on Engineering and Computer Science 2013 Vol I
WCECS 2013, 23-25 October, 2013, San Francisco, USA

Graph Theoretical Algorithms For JVM Operand Stack Visualization And Bytecode Verification

Sergej Alekseev, Andreas Karoly, Duc Thanh Nguyen and Sebastian Reschke

***Abstract*—The bytecode verification is an important task of the Java architecture that the JVM specification suggests. This paper presents graph theoretical algorithms and their implementation for the data flow analysis of Java bytecode. The algorithms mainly address the extended static visualization and verification of the JVMs operand stack to allow a deeper understanding in bytecode behavior. Compared to the well known algorithms, the focus of our approach is the visualization of the operand stack and a graph theoretical extension of the verification algorithms.**

We also show some experimental results to illustrate the effectiveness of our algorithms. All presented algorithms in this paper have been implemented in the Dr. Garbage tool suite. The Dr. Garbage project resulted from research work at the University of Oldenburg and is now further maintained at the University of Applied Sciences Frankfurt am Main. The tool suite is available for download under the Apache Open Source license.

***Index Terms*—java virtual machine, operand stack, verification, visualization, data flow analysis.**

I. INTRODUCTION

THE computational model of the Java Virtual Machine (JVM) corresponds to a stack machine [2]. Some other programming languages are also based on the computer model of a stack machine, for example *Forth* [4] and *PostScript* [3]. The algorithms and approaches presented in this paper are applicable to any stack based language, although we present our algorithms based on the JVM.

All bytecode instructions of the JVM take operands from the stack, operate on them and return results to the stack. Each method in a java class file has a stack frame. Each frame contains a last-in-first-out (LIFO) stack known as its operand stack [1, The Java® Virtual Machine Specification]. The stack frame of a method in the JVM holds the method's local variables and the method's operand stack. Although the sizes of the local variables get predetermined at the start of the method and always stay constant, the size of the operand stack dynamically changes as the method's byte code instructions are executed. The maximum depth of a frame's operand stack is determined at compile-time and is supplied along with the code for the method associated with the frame. Additionally, if a class is loaded by the JVM, the JVM verifies its content and makes sure there is no over- or underflow of the operand stack. But neither the Java compiler nor the JVM verifier perform a deep content analysis of the operand stack because such analyses are very time consuming and usually unnecessary, because the Java compiler generates reliable bytecode. Nevertheless, there are many tools that modify bytecode at runtime or generate it from different sources other than Java. In such cases, a more precise and detailed analysis of the operand stack is needed to localize potential runtime errors. The operand stack errors are very hard to track and most of them do not arise until many byte code instructions have already been executed.

In this paper we present graph theoretical algorithms for extended static verification and visualization of the operand stack. The algorithm *ASSIGN_OPSTACK_STATES* (section III-A) computes all possible contents of a method's operand stack. The following sections describe possible methods of analysis (size, type and content based) which can be performed on the calculated operand stack contents.

Section III-E describes a *LOOP_ANALYSIS* algorithm to handle the operand stacks of methods which contain cycles.

In section IV we propose a graph theoretical transformation algorithm to represent the operand stack structure and define a very simple grammar which includes the mathematical and logical operations in java similar syntax to visualize the contents and conditions of operand stacks based on the operand stack computation algorithm in section III-A.

All presented algorithms have been implemented in the context of the Dr. Garbage tool suite project [6] and we present some experimental results in section V which can be obtained from the Dr. Garbage tool suite project [6].

Furthermore, these algorithms are suitable as an extension of the Java compiler and JVM verifier.

II. RELATED WORK

Klein and Wildmoser explain improvements of Java bytecode verification in their papers [12, Verified lightweight bytecode verification], [14, Verified Bytecode Subroutines] and [13, Verified bytecode verifiers]. There the purpose of a verifier for bytecode regarding the Java operand stack and the possible misbehaviour like underflow and overflow are mentioned. The operand stack is shown as an array of types (e.g. int) per bytecode instruction. The described verification includes the type checking of operand stack entries. Stephen N. Freund and John C. Mitchell present in their paper [11, A Type System for the Java Bytecode Language and Verifier] a specification in the form of a type system for a subset of the bytecode language. And they developed a type checking algorithm and prototype bytecode verifier implementation.

The approach of Klein and Wildmoser, as well as the approach of Freund and Mitchell are partially related to our algorithm for the type based analysis in section III-C. But in addition to these algorithms we present a graph theoretical extension of the type based analysis.

Some other papers that deal with this subject are [17, Simple verification technique for complex Java bytecode

---

Manuscript received July 12, 2013; revised August 12, 2013.

Sergej Alekseev, Andreas Karoly and Duc Thanh Nguyen are with the Department of Computer Science, Fachhochschule Frankfurt am Main University of Applied Sciences, 60318 Frankfurt am Main, Germany (Phone. +49 69-1533-3673, e-mail: alekseev@fb2.fh-frankurt.de, karoly@stud.fh-frankfurt.de, thanh.nguyenduc1801@gmail.com).

Sebastian Reschke is with AVM GmbH, 10559 Berlin, Germany (e.mail: sebastian.reschke@drgarbage.com).

subroutine], [18, Java and the Java Virtual Machine - Definition, Verification, Validation], [19, A type system for Java bytecode subroutines] and [20, Subroutines and java bytecode verification].

Eva Rose deals in her paper [15, Lightweight Bytecode Verification] with the verification algorithms on embedded computing devices. Xavier Leroy's papers [9, Java bytecode verification: an overview] and [10, Java bytecode verification: algorithms and formalizations] review the various bytecode verification algorithms for crucial security Java components on the Web. The application field of our algorithm is not limited, but in view of the memory consumption (section III-A Algorithm) our approach in its pure form is not suitable for using on embedded devices.

In the paper [16, Analyzing Stack Flows to Compare Java Programs] of Lim and Han the Java operand stack is used by algorithms to identify clones of Java programs. They describe how the JVM specification defines the operand stack before and after each bytecode of a Java program. For visualization of the stack an array of dots with one dot per entry of the operand stack is used. Our approach provides a more versatile form of the operand stack representation.

III. STATIC OPERAND STACK ANALYSIS

An essential idea behind the static operand stack analysis is to identify a set of potential control flow paths in a method, to calculate all possible operand stack states by interpreting the byte code instructions for each path and to identify inconsistencies of the operand stack by comparing these states.

In the next subsection a graph theoretical algorithm is presented which traverses all control flow paths of a method and assigns the calculated operand stack states to each bytecode instruction in each control flow path.

The control flow path analysis is based on a control flow graph (CFG) of a method. The CFG is defined as a tuple $G = (V, A)$, where V is a nonempty set of *vertices* representing bytecode instructions of a method, A is a (possibly empty) set of *arcs* (or edges) representing transitions between the bytecode instructions. Formally A is the finite set of sets (a, b), where $a, b \in V$.

As a CFG containing loops has unlimited numbers of potential paths, the CFG has to be transformed into a directed acyclic graph (DAG) with a limited number of paths by removing loop backedges (as identified by a depth-first search of the CFG). The number of operand stack states in each acyclic path always equals the number of vertices (or number of corresponding byte code instructions) in this path.

The subsections III-B, III-C and III-D present techniques for a static operand stack analysis in acyclic graphs based on the operand stack size, type of stack variables and operand stack content. The subsection III-E extends operand stack analysis to arbitrary control flow graphs that contain cycles.

A. Algorithm for assigning stack states to vertices in a DAG

The algorithm *ASSIGN_OPSTACK_STATES* in fig. 1 identifies all possible control flow paths by visiting vertices of the DAG in topological order. This order ensures that all the predecessors of a vertex v are visited before v itself.

```
/* $G$ is directed acyclic graph defined as $G = (V, E)$ */
ASSIGN_OPSTACK_STATES ($G$)
1   for (each vertex $v \in V$ in topological order ){
2       $S[] = NULL$;
3       for (all incoming edges $l$ of $v$ ){
4           $v' = otherend(l, v)$;
5           $S = S \cup v'.stack$;
6       }
7       for (each stack state $s \in S$ ){
8           updateStack($s, v$);
9       }
10      v.stack = $S$;
11  }
```

Fig. 1. Algorithm for assigning stack states to vertices in a DAG

The algorithm calculates a list of all possible operand stack states for the current vertex v (fig. 1: lines 2-6) by iterating all the predecessors of the vertex v and building the set of stack states S as a disjunct union of all predecessors operand stack lists.

All stack states of the list S are updated by pop or push operations corresponding to the byte code instruction of the vertex v (fig. 1: lines 7-9).

After execution of the algorithm a list of all possible operand stack states is assigned to each vertex of the DAG.

Theorem 3.1: (Operand Stack Algorithm) Given a directed acyclic graph $G = (V, E)$, after the algorithm *ASSIGN_OPSTACK_STATES* in fig. 1 visits a vertex $v \in V$, the property variable $stack$ of v contains a list of all possible operand stack states in the vertex v.

Proof: By induction on the depth of a vertex $v \in V$ and paths from the $START$ vertex to v.

Base Case: v depth is 0 ($v = START$). The theorem is trivially satisfied.

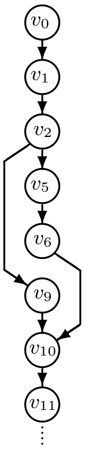
Induction Step: The list of all possible operand stack states S for the vertex v with $d > 0$ is calculated by updating all elements of the list $S = \{\forall s \in S | update(s)\}$ (lines 7-9). The list S is a set of all operand stack states of all immediate predecessors $v_0 ... v_n$ of v with the depth $d - 1$, so $\bigcup_{i=0}^{n} v_i.stack$ (lines 3-5). By induction hypothesis, each path section from the $START$ to v_i must be visited and all operand stack states are assigned to the property variable $v_i.stack$.

All successors $v_0 ... v_n$ of v must have a depth greater than d, because the graph is a DAG. So the theorem holds for all $v \in V$ with depth $d > 0$. ■

Fig. 3 illustrates how the algorithm operates on the example CFG. The vertices in this example are labeled in topological order. The following control paths exist:

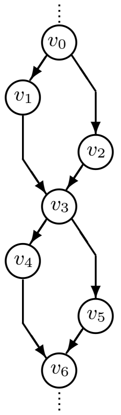
- $p_1 = \{..., v_0, v_1, v_3, v_4, v_6, ...\}$
- $p_2 = \{..., v_0, v_1, v_3, v_5, v_6, ...\}$
- $p_3 = \{..., v_0, v_2, v_3, v_4, v_6, ...\}$
- $p_4 = \{..., v_0, v_2, v_3, v_5, v_6, ...\}$

In path p_1 the vertex v_1 pushes the variable a and the vertex v_4 pushes the variable c onto the stack. The operand stack states can be assigned to each vertex of path p_1 as follows: $p_1 = \{..., v_0(-), v_1(a), v_3(a), v_4(a, c), v_6(a, c), ...\}$. According to these steps, the stack states in all paths can be calculated and assigned to the vertices in the DAG. But this



Byte code	Stack size	List of variable types	List of variables on the stack
0 iload_1;	1	I	a
1 iload_2;	2	I,I	a, b
2 if_icmple 7;	0	-	-
5 iload_1;	1	I	a
6 goto 4;	1	I	a
9 iload_2;	1	I	b
10 iconst_2;	2	I,I	a, 2 | b, 2
11 invokestatic 16;	0	-	-

Fig. 2. CFG, corresponding byte code and the operand stack representation.



Stack operation	Stack states
-	-
push a;	a
push b;	b
-	a | b
push c;	a, c | b, c
push d;	a, d | b, d
-	a, c | a, d | b, c | b, d

Fig. 3. CFG with operand stack states computed by the algorithm.

procedure is not efficiently in terms of runtime complexity. To calculate all possible stack states in each vertex of a DAG it is not necessary to traverse each control path separately. Instead our algorithm calculates the stack states step by step for all paths by visiting the vertices of a DAG in topological order.

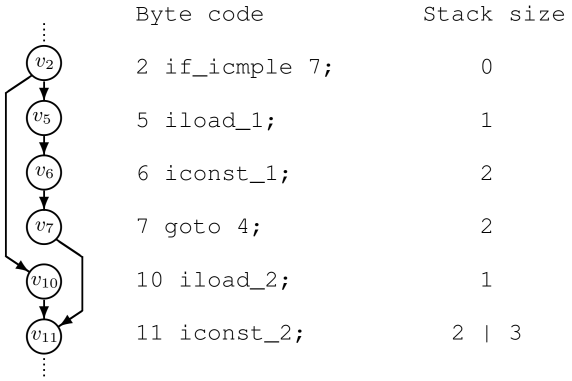
Generally, the runtime complexity of a topological search algorithm for the given directed acyclic graph G with n vertices and m arcs can be found in $O(n+m)$ (see [7] or [8]). The memory allocation complexity to store all possible operand stack combinations in our algorithm grows exponentially. As you can see from the example in fig. 3 the number of combinations N depends on the number of sequential branches in the DAG and equals the multiplication of the number of branches in each branch. In this case $N = 2 \times 2 = 4$. So the complexity of the memory allocation can be calculated as $O(n^n)$. To solve this problem a pragmatic approach is used in our implementation. We define the maximum number of combinations which have to be calculated by the algorithm to limit the memory allocation. The number of maximum combinations is variable and can be redefined for each operand stack.

The algorithm *ASSIGN_OPSTACK_STATES* in fig. 1 can be easily adapted to calculate the stack depth (used for size based analysis) and the list of variable types (used for type based analysis) in each node. Instead of the operand stack state combinations, a single value is stored in the property variable $stack$ of each vertex. In this case, both the runtime $O(n+m)$ and the memory allocation $O(n)$ have linear complexity. An example of the operand stack representation is illustrated in the fig. 2.

B. Size based operand stack analysis

A size based analysis can be achieved by simply altering the algorithm *ASSIGN_OPSTACK_STATES* in fig. 1 to calculate the operand stack depth value and store it in the property variable for each vertex in the corresponding CFG. By trivial comparison of the operand stack depth values assigned to the CFG's vertices, the following types of inconsistencies can be determined:

- **Stack over or underflow:** The max operand stack size is calculated as the algorithm visits the vertices of the corresponding CFG in topological order. By comparing the calculated max size with the max stack size, stored in the class file, over- or underflow stack errors can be determined. The overflow verification is generally available in the JVM as specified in [1, The Java® Virtual Machine Specification]. Our approach also allows to determine the bytecode addresses of the instructions which cause the stack overflow.
- **Leaving objects on stack:**
  The stack size of each end vertex (e.g. return byte instruction) in the CFG is verified. Herewith is determined if any objects remain on the stack. By the reference to the bytecode instructions of the remained objects a warning is generated about possibly unused byte code instructions (instructions which push these objects onto the stack).
- **Asymmetrical operand stack sizes:** An error in one branch of the CFG could lead to asymmetrical operand stack sizes on the incoming edges of a vertex as illustrated in fig. 4. A simple backtrace algorithm to find unused instructions, is applied in our implementation. A more complex analysis algorithm and a backtrace implementation is planned for the future.



Byte code	Stack size
`2 if_icmple 7;`	0
`5 iload_1;`	1
`6 iconst_1;`	2
`7 goto 4;`	2
`10 iload_2;`	1
`11 iconst_2;`	2 | 3

Fig. 4. Asymmetrical operand stack size inconsistency

The runtime complexity for this analysis is in $O(n + m)$ and the memory allocation complexity is in $O(n)$, where n is the number of vertices and m number of arcs in the CFG.

C. Type based operand stack analysis

According to the Java® Virtual Machine Specification [1] the JVM supports the operand stack type verification in general. Gerwin Klein and Tobias Nipkow formalize and describe algorithms for an iterative data flow analysis that statically predicts the types of values on the operand stack and in the register set [14], [12], [13] as mentioned in section II. In this section we present a graph-theoretical approach in addition to the well known verification techniques.

A type based analysis is realized by adaptation of the algorithm *ASSIGN_OPSTACK_STATES* in fig. 1 to calculate a list of variable types on the stack and store it in the property variable for each vertex in the corresponding CFG. By comparison of the values assigned to the CFG's vertices the following types of inconsistencies can be determined:

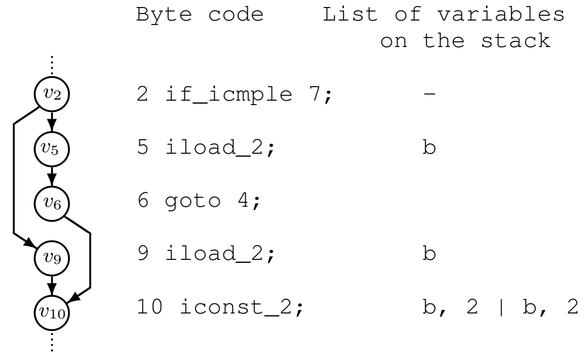
- **Expected type:** To ensure proper code execution at runtime, all operands on the stack have to be type correct in terms of what operand type the bytecode instruction expects. For example an `istore` instruction can not handle a `float` operand.
- **Asymmetrical type lists:** The types of operands on a stack can differ on the incoming edges of a vertex. The backtrace algorithm allows to reference the bytecode instructions which pushed operands with different types onto the stack.

The runtime complexity for this analysis is in $O(n + m)$ and the memory allocation complexity is in $O(n)$, where n is the number of vertices and m number of arcs in the CFG.

D. Content based operand stack analysis

The algorithm *ASSIGN_OPSTACK_STATES* in fig. 1 calculates a list of variables on the stack for each bytecode instruction and stores it in the property variable of the vertex in the corresponding CFG. In a certain vertex several variable combinations on the stack are possible.

This analysis allows to figure out unnecessary branches in the bytecode. The bytecode example in fig. 5 contains an if-branch. The bytecode instructions (offset `5` and `9`) in both branches push the same variable `b` onto the stack. A backtrace algorithm prints the bytecode addresses of instructions which lead to the duplicated operand stack states.



Byte code	List of variables on the stack
`2 if_icmple 7;`	-
`5 iload_2;`	b
`6 goto 4;`	
`9 iload_2;`	b
`10 iconst_2;`	b, 2 | b, 2

Fig. 5. Operand Stack with the same content in two branches

This kind of analysis is related to compiler optimization techniques, but in our approach the operand stack analysis is used to localize unused instructions. Our approach is partially comparable to the method of Lim and Han described in their paper [16, Analyzing Stack Flows to Compare Java Programs]. Although the goal of their paper is to identify clones of Java programs, the approach is absolutely different.

The runtime complexity for this analysis is in $O(n+m)$, where n is the number of vertices and m number of arcs in the CFG. The memory allocation complexity is in $O(n^n)$ as mentioned in the section III-A.

E. Loop based operand stack analysis

This section extends the analysis algorithms to arbitrary control flow graphs that can contain cycles. The algorithm fig. 1 in section III-A only works for acyclic paths, which correspond to back edge free paths. The main idea of the loop based analysis is that the operand stack states before entering and after leaving a loop have to be equal. Otherwise, each iteration of the loop would push objects onto the stack or pop them from the stack and the state of the stack would be undefined.

A depth-first search algorithm identifies a set of back edges $B \subset E$ in a graph $G = (V, E)$, that contains cycles. The graph G is transformed into a directed acyclic graph (DAG) by removing the back edges $D = (V, E')$, where $E' = E \backslash B$. Each back edge $b \in B$ lies on a loop.

Theorem 3.2: (Loop Analysis Algorithm) The operand stack of a method represented by a control flow graph G that contains cycles is consistent if:

1) Size based analysis (section III-B) and type based analysis (section III-C) have been performed without any error on the directed acyclic graph D transformed from the graph G.
2) and for each back edge $b \in B$ with the start vertex $v_s \in V$ and the end vertex $v_e \in V$: the operand stack state assigned to the start vertex v_s and the states assigned to the start vertices $v_0, ...v_n$ of all incoming edges of the end vertex v_e are equal in size and type.

Proof: The point 1 of the theorem does not need to be proofed, because in case of any errors the stack is inconsistent. The point 2 can be proofed by the contradiction of the operand stack states.

Let us consider the directed acyclic graph D produced by removing the back edges and the set of back edges B. For each $b \in B$ holds:

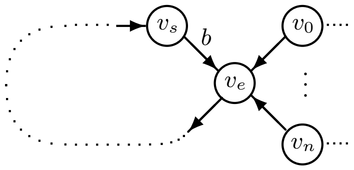


Fig. 6. Loop based analysis

- Each back edge $b \in B$ lies on one loop.
- There are possibly several forward paths $v_e \to ... \to v_s \to v_e$ in the loop.

Each forward path must have the same operand stack state in the last vertex, because all paths are acyclic and they pass the back edge b. All acyclic paths have already been verified by the point 1 of the theorem.

The back edge b is a single back edge of the vertex v_s, because all other edges are outside the loop and must belong to the DAG. So all states of other incoming edges have been already verified by the point 1 of the theorem.

If the state of the vertex v_e would not be equal to the states of vertices $v_0 ... v_n$ then the stack would be inconsistent. ■

The following algorithm for the loop based analysis is derived from the theorem 3.2. The algorithm executes the operand stack comparison for all back edges $b \in B$ identified in the previous step of the analysis. The runtime complexity for this analysis is in $O(n+m)$ where n is the number of vertices and m number of arcs in the CFG.

```
/* D is a directed acyclic graph, $D = (V, E')$. B is a   */
/* set of back edges, $B \nsubseteq E'$. The back edge $b \in B$,   */
/* $b = \{v_s, v_e\}$, where $v_s, v_e \in V$.              */
LOOP_ANALYSIS (D, B)
    for (each back edge $b \in B$ ){
        for (all incoming edges $l$ of $v_e$ ){
            $v = otherend(l, v_e)$;
            if($v.stack \neq v_s.stack$) {
                print ERROR;
    }   }   }
```

Fig. 7. Loop based analysis algorithm

IV. VISUALIZATION OF THE OPERAND STACK

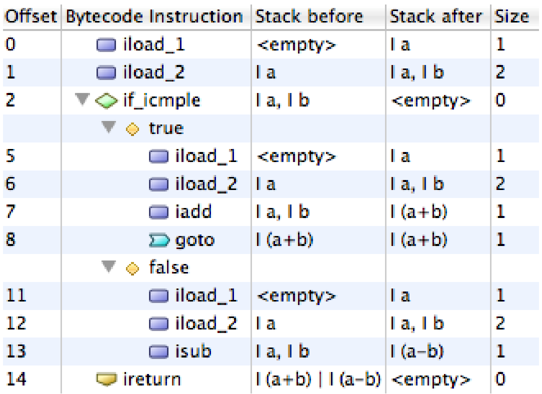
The simplest way to visualize the operand stack is to calculate the state of the operand stack in each instruction of a method and display the complete list of instructions with corresponding states. The calculation of the operand stack states is performed by the algorithm *ASSIGN_OPSTACK_STATES* in section III-A fig. 1. The simple representation of the operand stack is shown in fig. 2, section III-A. To make the operand stack content more comprehensible we have defined a grammar (see appendix A) which includes variable types and names, logical combination and arithmetical operations and developed an algorithm to transform the control flow graph of a method to a tree representation.

The algorithm *TRANSFORM_GRAPH* in fig. 8 creates a basic block graph G_b for the given control flow graph G (line 1), removes the back edges in G_b (line 2) and starts a *Depth First Search* from each start basic block B, where $indegree(B) = 0$ (line 4). For each basic block B a new tree node n is created (Routine *CREATE_TREE: line 1*) and all vertices of a basic block B (each vertex represents a bytecode instruction) are added as children of B to the tree T (Routine *CREATE_TREE: lines 2-4*). The bytecode instruction tree T is used to represent the operand stack structure in a view.

```
/* G is a control flow graph, $G = (V, E)$.                  */
TRANSFORM_GRAPH (G)
1   $G_b = createBasicBlockGraph(G)$;
2   $removeBackEdges(G_b)$;
3   T = createTree();          /* T is an empty tree. */
4   for (all start basic blocks $B \in V_b$)
5       $CREATE\_TREE(B, T)$;
6   };

CREATE_TREE (B, T)
1   add a new tree node n for B to T;
2   for ( all vertices $v \in B$ ){
3       add v as child of n to the tree T;
4   }
5   for (all outgoing args l of B ){
6       $CREATE\_TREE(otherend(l, B), T)$;
7   }
```

Fig. 8. Transformation algorithm for the operand stack representation



Offset	Bytecode Instruction	Stack before	Stack after	Size
0	iload_1	<empty>	I a	1
1	iload_2	I a	I a, I b	2
2	▼ if_icmple	I a, I b	<empty>	0
	▼ true			
5	iload_1	<empty>	I a	1
6	iload_2	I a	I a, I b	2
7	iadd	I a, I b	I (a+b)	1
8	goto	I (a+b)	I (a+b)	1
	▼ false			
11	iload_1	<empty>	I a	1
12	iload_2	I a	I a, I b	2
13	isub	I a, I b	I (a–b)	1
14	ireturn	I (a+b) | I (a–b)	<empty>	0

Fig. 9. Operand stack representation

An example representation of a Java bytecode is shown in fig. 9.

V. EXPERIMENTAL RESULTS

As stated in section III-A the more combinations of sequential branches are contained in the bytecode of a method, the more memory needs to be allocated. In practice, excessive memory allocation happens very rarely. We analyzed over 500 methods from different Java classes of the Standard Java Library with the implementation based on the Dr. Garbage tools [6], [5]. The five most representative methods are listed in the table I which have been selected by the following criteria:

- methods with a large number of bytecode instructions
- methods that contain a large number of `if` or `switch` instructions
- methods that hold a decent amount of stacks

TABLE I
EXPERIMENTAL RESULTS: MS - MAX STACK SIZE, C - MAX NUMBER OF STACK COMBINATIONS, AS - # OF ASSIGNED STACKS, SE - # OF STACK ENTRIES, BI - # OF BYTE CODE INSTRUCTIONS, IF/S - # OF IF/SWITCH INSTRUCTIONS

Package Path (class.method)	MS	C	AS	SE	BI	IF/S
sun.awt.geom.Curve.compareTo()	31	1	1	18	508	25/0
XmlChars.isCompatibilityChar()	2	2	1	2	309	85/1
Font3D.triangulateGlyphs()	6	2	2	5	2462	84/0
Matrix3d.compute_svd()	10	1	1	8	1558	19/0
javax.media.j3d.Alpha.value()	5	1	1	5	775	39/0

The experimental results have shown that despite a number of conditional branch operators or stack entries along with method instructions, the amount of stack combinations stay in limit.

VI. CONCLUSION

This paper describes new algorithms for operand stack analysis and visualization based on graph theoretical methods. Although the algorithms partially execute trivial operand stack verifications, they can be obtained as a supplement to the well known algorithms. The operand stack visualization algorithms presented in this paper are the first that can represent the operand stack in such a comprehensive way.

Experimental results showed that the performance and memory consumption do never deviate from linearity, although the theoretical memory consumption has exponential complexity. It is obviously possible with the synthetically generated code to reach the limits, but such code constructs do not occur in practice. We are convinced that a lot of new tools can be designed and implemented based on these algorithms and results

APPENDIX A
OPERAND STACK CONTENT GRAMMAR

```
<Stack> ::= <StackEntry>{"," <StackEntry>}{"|" <Stack>}
<StackEntry> ::= <type> <value>
<type> ::= "B" | "C" | "D" | "F" | "I" | "J" | "S" | "Z"
         | "L" | <array type>
<array type> ::= "[" { "[" }  <type>
<value> ::= <variable name> | <constant> | <array name>
          | <math operation>
<variable name> ::= <char>{<char>}
<char>::= any one of the 128 ASCII characters, but not
any of special characters [, ], (, ), \, /,  ;, ... or
the space character
<constant> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
                <float_constant>  {<constant>}
<array name> ::= <variable name> [ <numeric> |
                 <variable name> ]
<float_constant> ::= <constant> "."  <constant>
<math operation> ::="("<value>  <operation>  <value>")"
<operation> ::= "+" | "-" | "*" | "/" | "%" | "^" | "|"
              | "<<" | ">>"
```

ACKNOWLEDGMENT

The authors would like to thank the Dr. Garbage Project Community for supporting the implementation of proposed algorithms and tools [5].

REFERENCES

[1] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley, *The Java® Virtual Machine Specification*, Java SE 7 ed., 2013, http://docs.oracle.com/javase/specs/jvms/se7/html/
[2] Bruce Ian Mills, *Theoretical Introduction to Programming*, Springer 2006, ISBN 978-1-84628-263-8
[3] Adobe Systems, *PostScript language reference. 3.Edition* , Addison-Wesley 1999, ISBN 0-201-37922-8
[4] Leo Brodie, *Starting FORTH: an introduction to the FORTH language and operating system for beginners and professionals*, Prentice-Hall 1987, ISBN 0-201-37922-8
[5] Sergej Alekseev, Peter Palaga and Sebastian Reschke, *The Dr. Garbage Tools Project*, 2013, http://www.drgarbage.com
[6] Sergej Alekseev, Victor Dhanraj, Sebastian Reschke, and Peter Palaga, *Tools for Control Flow Analysis of Java Code*, Proceedings of *the 16th IASTED International Conference on Software Engineering and Applications*, 2012, http://www.actapress.com/PaperInfo.aspx?paperId=454811
[7] Gnther Stiege, *Graphen und Graphalgorithmen*, Shaker; *Auflage: 1*, 2006, ISBN 3832251138
[8] Donald E. Knuth, *The Art of Computer Programming*, Addison Wesley, 1997, ISBN 0201896834
[9] Xavier Leroy, *Java bytecode verification: an overview*, Computer Aided Verification, CAV 2001, Vol. 2102 of Lecture Notes in Computer Science, pages 265-285. Springer, 2001.
[10] Xavier Leroy, *Java bytecode verification: algorithms and formalizations*, Journal of Automated Reasoning, Vol. 30 Issue 3-4, Pages 235 - 269, 2003, http://gallium.inria.fr/ xleroy/publi/bytecode-verification-JAR.pdf
[11] Stephen N. Freund, John C. Mitchell, *A Type System for the Java Bytecode Language and Verifier*, Journal of Automated Reasoning, Vol. 30 Issue 3-4, Pages 271 - 321, 2003, http://theory.stanford.edu/people/jcm/papers/03-jar.pdf
[12] Gerwin Klein, Tobias Nipkow, *Verified lightweight bytecode verification*, Concurrency and Computation: Practice and Experience, Vol. 13, Pages 1133-1151, 2001
[13] Gerwin Klein, Tobias Nipkow, *Verified bytecode verifiers*, Journal Theoretical Computer Science, Vol. 298, Issue 3, Pages 583 - 626, 2003, http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2003/klein.pdf
[14] Gerwin Klein, Martin Wildmoser, *Verified Bytecode Subroutines*, Journal of Automated Reasoning, Vol. 30 Issue 3-4, Pages 363 - 398, 2003, http://www.cse.unsw.edu.au/ kleing/papers/KleinW-TPHOLS03.pdf
[15] Eva Rose, *Lightweight Bytecode Verification*, Journal of Automated Reasoning, Vol. 31, Issue 3-4, Pages 303-334, 2003
[16] Hyun-il Lim, Taisook Han, *Analyzing Stack Flows to Compare Java Programs*, EICE Transactions 95-D(2), Pages 565-576, 2012
[17] Alessandro Coglio, *Simple verification technique for complex Java bytecode subroutine*, In Proc. 4th ECOOP Workshop on Formal Techniques for Java-like Programs, 2002
[18] Robert Strk, Joachim Schmid, and Egon Brger, *Java and the Java Virtual Machine - Definition, Verification, Validation*, Springer, 2001.
[19] R. Stata and M. Abadi, *A type system for Java bytecode subroutines*, In Proc. 25th ACM Symp. Principles of Programming Languages, Pages 149–161. ACM Press, 1998.
[20] Martin Wildmoser, *Subroutines and java bytecode verification*, Master's thesis, Technische Universitt Mnchen, 2002.