Decompiling Boolean Expressions from Java™ Bytecode

Mangala Gowri Nanda
IBM Research, India
mgowri@in.ibm.com

S. Arun-Kumar
Indian Institute of Technology, Delhi
sak@cse.iitd.ernet.in

ABSTRACT

Java bytecode obfuscates the original structure of a Java expression in the source code. So a simple expression such as (c1 `||` c2) or (c1 `&&` c2) may be captured in the bytecode in 4 different ways (as shown in the paper). And correspondingly, when we reconvert the bytecode back into Java source code, there are four different ways this may happen. Further, although `goto`s are not permitted in the Java source code, the bytecode is full of `goto`s. If you were to blindly convert the bytecode into Java source code, then you would replace a `goto` by a labeled break. A labeled break has the advantage that it only allows you to break out of a block structure and (unlike a setjump) does not permit you to jump arbitrarily **into** a block structure. So while the data structures used in the regenerated Java source code are still relatively "clean" arbitrary usage of labeled breaks makes for unreadable code (as we show in the paper). And this can be a point of concern, since decompilation is generally related to debugging code.

Instead of dumping arbitrary labeled breaks, we try to reconstruct the original expression, in terms of `&&` and `||` clauses as well as ternary operators "?:" (c0 ? c1 : c2); Thus our goal is quite simply to regenerate, without using `goto` or labeled breaks, the expressions as close to the original as possible (it is not possible to guarantee an exact match). In this paper we explain what is the state of the art in Java decompilers for decoding complex expressions. Then we will present our solution. We have implemented the algorithms described here in this paper and give you our experience with it.

CCS Concepts

•**Social and professional topics → Software reverse engineering;**

Keywords

decompilation; Java bytecode; boolean expressions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISEC '16, February 18-20, 2016, Goa, India

© 2016 ACM. ISBN 978-1-4503-4018-2/16/02…$15.00

DOI: http://dx.doi.org/10.1145/2856636.2856651

1. INTRODUCTION

Our motivation to work on this problem stemmed from the problem of generating executable code from a slice of a program. The trouble with working with Java bytecode is that the bytecode does not preserve the block structure of the original program. So, for example, it becomes very hard to figure out where to put the curly braces in the newly reconstituted program. There are essentially 3 issues related to regenerating Java code from bytecode:

1. Figuring out where to put the curly braces.

2. Converting `goto` in the bytecode into appropriate continue and break statements.

3. Regenerating complex boolean expressions without using `goto` statements.

When dealing with sliced code, we have the additional problem of determining which `goto` to include in the slice.

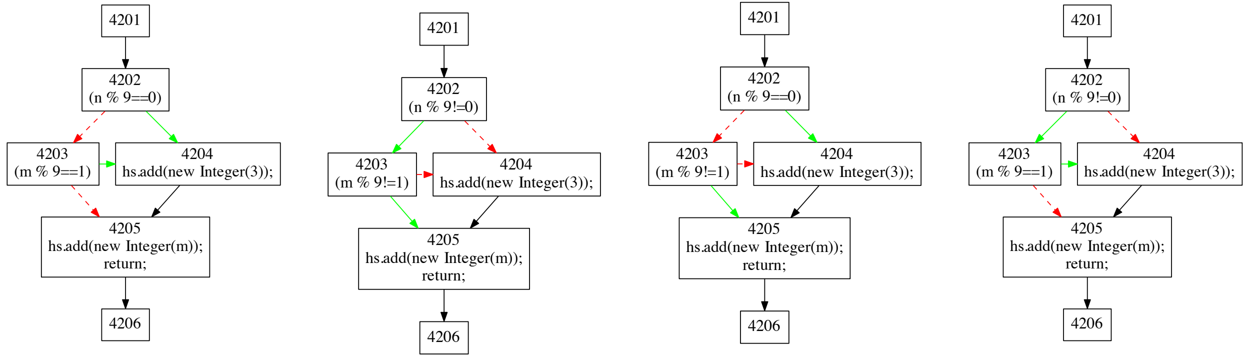
Due to lack of space, in this paper, we concentrate only on the problem of regenerating complex boolean expressions.

Problem Definition. A generic problem related to decompiling code from a control flow graph is that it is difficult to regenerate code when there are one or more `OR` or `AND` clauses in a conditional.

A Brief look at Boolean Expressions. It turns out that recognizing and generating code for boolean expressions with `OR` clauses (c1 `||` c2) and `AND` clauses (c1 `&&` c2) is not easy. To this we add the boolean expressions (c0 ? c1 : c2) and the boolean expression (c0 ? a1 : a2)==val3 and things are definitely beginning to get more complex. Furthermore we note that any of the conditionals, c0, c1 or c2, in the preceding expressions could be replaced by any of the preceding expressions. For example, if we were to take the expression (c0 ? c1 : c2) and replace c0 by an `OR` clause (x1 `||` x2), replace c1 by an `AND` clause (x3 `&&` x4) and replace c2 by the boolean expression (x5 ? a1 : a2)==val3, then we would get the boolean expression

$$(x1 \texttt{ || } x2)?(x3 \texttt{ \&\& } x4) : ((x5?a1 : a2) == val3)$$

Now each of the conditionals can further be replaced by another set of complex boolean expressions (including the one that we just generated). And so on, ad infinitum. Thus, as you can see, expressions can become very complex very fast.



(a) The Classic `OR`

```
if(n % 9==0) goto 4204
    else goto 4203
4203: if(m % 9==1) goto 4204
          else goto 4205

4204: hs.add(new Integer(3));
goto 4205

4205: hs.add(new Integer(m));
return;
```

(b) The `AND` equivalent

```
if(n % 9!=0) {
  if(m % 9!=1) {
  }
  else {
    hs.add(new Integer(3));
  }
}
hs.add(new Integer(m));
return;
```

(c) An alternative

```
if(n % 9==0) goto 4204
    else goto 4203
4203: if(m % 9!=1) goto 4205
          else goto 4204

4204: hs.add(new Integer(3));
goto 4205

4205: hs.add(new Integer(m));
return;
```

(d) Another alternative

```
if(n % 9!=0) goto 4203
    else goto 4204
4203: if(m % 9==1) goto 4204
          else goto 4205

4204: hs.add(new Integer(3));
goto 4205

4205: hs.add(new Integer(m));
return;
```

Figure 1: Equivalent CFGs for a simple `OR` clause along with the corresponding code.

A Very Simple Boolean Expressions. Consider the simple expression shown in the code below. There are just two clauses (or sub-expressions) joined with a single ||.

```
if ( n % 9 == 0 || m % 9 == 1 ) {
  hs.add(new Integer(3));
}
hs.add(new Integer(m));
```

Let us observe what kind of Control Flow Graph (CFG) may get generated for this code. The classic `OR` pattern is shown in Figure 1(a). The classic `AND` version of the `OR` clause is shown in Figure 1(b), where both the conditionals have been reversed and so have the edges. However, there are two more equivalent representations of the same conditional shown in Figure 1(c) where only the conditional at N_{4202} is reversed (along with the corresponding edges) and in Figure 1(d) where only the conditional at N_{4203} is reversed (along with its corresponding edges).

NOTE: All four CFGs are *equivalent*. Depending on the compiler and the bytecode generator, any one of these four patterns may be generated and we would need to recognize either the `OR` pattern or the `AND` pattern in order to generate code. As a matter of fact, the WALA [4] bytecode analysis engine generates edges as shown in Figure 1(d).

This problem is further compounded when there is more than one `OR` or `AND` clause or if there is a combination of `OR` and `AND` clauses.

This issue would have become a non-issue had we been permitted to use `goto` statements and we could then have generated code as shown in the Figures 1(a), (c) and (d). Figure 1(b) is the only version where we are able to generate regular code without `goto`s. We observe that in a CFG if there are two (or more) `true edges`, and / or `false edges` directed at the same `basic-block`, then it is not possible to generate code without using `goto` statements—unless, we can reduce the expression to an equivalent expression using `OR` and `AND` clauses. Our code generation algorithm is explained in Section 3.

Using Labeled breaks.

Meanwhile let us take a sneak peek at what other people do. First we look at SOOT. SOOT [15] is the antithesis of all that we are working towards. SOOT generates by converting the `goto` into labeled break statements. This makes code generation much simpler, but makes the code unreadable for human beings. For the simple example given above, SOOT generates the following code:

```
label_32: {
    if (n % 9 != 0 && m % 9 != 1) {
        break label_32;
    }
    hs.add(new Integer(3));
} //end label_32:

hs.add(new Integer(m));
```

So while they do use labeled breaks, the code is still readable.

However, even some extremely simple code such as one involving ternary booleans

```
if ( c0 ? c1 : c2 ) {
   sTrue();
 }
 else {
   sFalse();
 }
```

turns into the following incomprehensible 17 lines of code.

```
label_14: {
    label_13: {
        if ( ! (z0)) {
            if ( ! (z2)) {
                break label_13;
            }
        }
        else {
            if ( ! (z1)) {
                break label_13;
            }
        }
        sTrue();
            break label_14;
    } //end label_13:
    sFalse();
} //end label_14:
```

And instead of the compact 1 line of code they generate 72 lines of (humanly unreadable) code for the following code:

```
if(X?(W?G:(V?G:R)):(U?(V?G:R):R)){
    sTrue();
}
else {
  sFalse();
}
```

In addition, we investigated a new breed of freely available Java decompilers, including JODE [7], JREVERSEPRO [9], PROCYON [14], FERNFLOWER [13] and CFR [2]. While most of them generate correct compilable code on all of our sample programs, there was not one of them that could handle ternary expressions "elegantly". They all get confused and (like SOOT) generate some fairly convoluted (albeit correct) code when given some complex expressions involving ternary expressions. (In Section 4 we give you a comparison of the various tools.) In this paper we tackle the problem of generating "human readable" code from Java bytecode for boolean conditional expressions.

Problem statement.

We are now ready to define the problem statement as follows:

To regenerate Java source code for all boolean expressions from Java bytecode without using any `goto` or labeled break statements.

Contributions.

The contributions of this paper are

- An algorithm to regenerate conditional expressions from bytecode without using `goto` or labeled break statements either
- An empirical evaluation of the efficiency and usability of our algorithms, along with a comparison with other tools.

2. PRELIMINARIES

In this section we describe the preliminary analysis that needs to be done before we can start the code generation.

The *control-flow graph* (CFG) for a method M contains *nodes* that represent statements in M and *edges* that represent potential flow of control among the statements. It has a unique `Entry` node from which all other nodes are reachable and a unique `Exit` node that is reachable from all other nodes.

2.1 Setting up the Control Flow Graph

The first step in decompiling is to remove all exception flow edges, except those that arise from an explicit `throw` command. Then we need to merge all straight line code into single `basic-blocks`. That is, each `basic-block` either ends with a conditional (that is, it has two out-going edges, one marked `true` and the other marked `false`) or the succeeding `basic-block` has more than one incoming edge.

DEFINITION 1. Dominator*: A node* S_i dominates *a node* S_j iff $S_i \neq S_j$ *and* S_i *is on every path from* `Entry` *to* S_j.

DEFINITION 2. Backedge*: A backedge in the CFG, is an edge where the destination of the edge dominates the source of the edge.*

We compute the dominators and find the `back edges` in the graph. Then we traverse the CFG in depth first order (ignoring the `back edges`) to generate the `basic-blocks` in topological order [1]. All our operations are conducted on these two data structures, the CFG and the topological order listing of the `basic-blocks`. Both data structures consist of the same set of `basic-blocks`.

- The CFG (the original data structure) consists of the set of `basic-blocks` with control flow edges. Each edge has an attribute indicating whether it is a `true edge` or a `false edge`. A `conditional-block` is the source of exactly one `true edge` and exactly one `false edge`) and each `true edge` and `false edge` originates only at a `conditional-block`.
- The topological order[1] list consists of an ordered set of the same `basic-blocks`, such that (ignoring `back edges`) a `basic-block` in the list precedes any (transitive) successor in the CFG.

  When we generate the topological order list, we also ensure that at a `conditional-block`, the destination of a `true edge` always precedes the destination of a `false edge`. More about this in Section 3.1.

In the rest of this paper, we use the terms node and `basic-block` interchangeably.

2.2 The Topological order

Since the topological order is such an important concept in this paper we decided to give you some more details about it.

There is an exponential number of ways in which we can write the code of a program.

For example we could write	or we could write

`if ( cond ) {`
`{ ... then statements ...}`
`}`
`else {`
`{ ... else statements ...}`
`}`
	`if ( !cond ) {`
	`{ ... else statements ...}`
	`}`
	`else {`
	`{ ... then statements ...}`
	`}`

It really does not matter. The topological order of nodes will

[1]Note that a topological order is *not* the same as a preorder or a breadth-first order

generate the nodes in any one such sequence. So a traversal in topological order can output the code in the some prescribed sequential order. The algorithm is given in Algorithm 1.

Algorithm 1 The algorithm for generating the Topological Order from a CFG.

```
function getBlksinTopolgclOrdr0()
    entry ← getEntryBlock
    blks ← new Vector
    seen ← new HashSet
    dfslist ← new HashSet
    getBlksinTopolgclOrdr1(entry, dfslist, seen, blks)
    return blks
end function
function getBlksinTopolgclOrdr1(bb, dfslist, seen, blks)
    if seen.contains(bb) then
        return
    end if
    seen.add(bb)
    dfslist.add(bb)
    for all ss ∈ bb.getSuccessors() do
        if !dfslist.contains(ss) then
            getBlksinTopolgclOrdr1(ss, dfslist, seen, blks)
        end if
    end for
    dfslist.remove(bb)
    blks.insertElementAt(bb, 0)
end function
```

2.3 Setting up the bytecode

The bytecode for a statement of the form `a.b.c = d` is broken down as follows: `t1 = a.b`, `t1.c = d` and similarly, a statement of the form `d = a.b.c` is broken down as `t2 = a.b`, `d = t2.c`. Similarly expressions of the form `n = x+y*z-k` would have been broken down into the subexpressions `t3 = y * z`, `t4 = x + t3`, `n = t4 - k`. Also a function call `foo(a.b.c, x+y*z-k)` would have been converted into the call `foo(d, n)`.

All of these statements now need to be reconstituted. This is to ensure that a statement of the type `if (b.f.g == foo(a.b.c, x+y*z-k) && c.equals(b))` can be recaptured correctly.

This is a fairly straight forward algorithm, where we walk through the code in each basic block doing the following:

- for each statement that has an "=" in it (whether it is a `GetField`, `Putfield`, `BinaryOp`, `UnaryOp`, `assignment`, `Invoke`, etc.) we capture the string value of the RHS and store it with the LHS.
- Then any place the LHS is used, it is replaced by the RHS string value and the corresponding assignment statement is deleted.

3. CODE GENERATION

Code generation from a control flow graph may sound like a trivial problem. However it is not quite so easy when one is not permitted to use `goto` statements. This creates problems even though the source program itself does not permit arbitrary `goto`s. While there exist several tools that claim to decompile Java from bytecode, only some of them worked correctly on all our sample programs. Yet, although `Jode`, `Procyon` and `FernFlower` do generate correct code they also do use labeled break statements in certain cases. Finally, we did not find any documentation on decompilation and so we give here an algorithm that decompiles Java without resorting to the use of `goto` statements.

3.1 Handling clauses in a conditional

The edges emanating from various nodes of the CFG may be classified into `true edges` (green), `false edges` (red) and `unconditional edges` (black). A node which represents a condition may have exactly one `true edge` and one `false edge` emanating from it.

The problem with decompiling conditionals is that when we have two or more incoming edges at a given node and at least one of them is a `true edge` or a `false edge`, then it is not possible to generate code without `goto`s—unless we can combine two or more conditionals using some combination of || and && clauses in such a way that the `true edges` / `false edges` get absorbed by the process of the combination.

In the rest of this paper, we use the term `green edges` and `true edges` interchangeably, and similarly we use the term `red edges` and `false edges` interchangeably.

Determining an appropriate boolean expression from the given graph hinges on the following theorem, which we term the monochromatic theorem:

THEOREM 1. *For each `basic block` that participates in a boolean expression, all the incoming edges must be the same color. That is, all incoming edges are either `true edges` or they are `false edges`, or in the case of certain ternary clauses, they may be unconditional edges (`black edges`).*

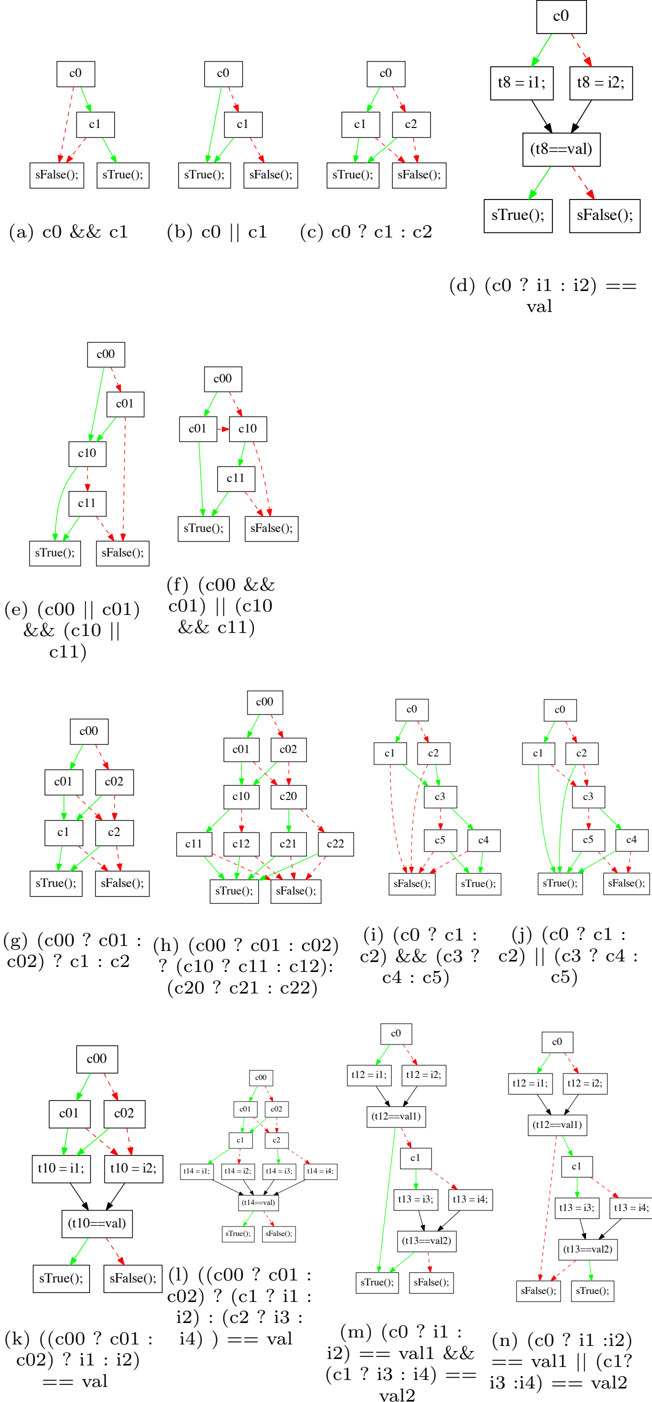
In order to prove the theorem, it is convenient to restrict ourselves to a maximal directed-acyclic subgraph (DAsG) of the CFG which satisfies the following conditions.

1. there is a single entry node (called the root) into the subgraph such that all incoming edges are unconditional,
2. every node in the subgraph is an atomic condition,
3. every incoming edge into a non-root node is from some other node in the same DAsG.
4. every exit from every node of the DAsG is either to another condition in the DAsG or to one of two other basic blocks S_{true} and S_{false} in the CFG

Each such subgraph represents a (possibly complex) condition of a branching or looping statement with control flowing into it through the root-node and out to the nodes S_{true} and S_{false} [2]. The proof of the theorem then follows easily from lemma 1.

[2]Note however that

- There could be unconditional edges into S_{true} and S_{false} from outside this subgraph too. Since the subgraph consists of nodes which are all conditions each exit from the subgraph is either a *true edge* or a *false edge*.
- If there are unconditional edges coming into the non-root nodes of the DAsG then one must consider the subgraph as representing not a single monolithic boolean expression but as one representing a nested



(a) c0 && c1 (b) c0 || c1 (c) c0 ? c1 : c2 (d) (c0 ? i1 : i2) == val

(e) (c00 || c01) && (c10 || c11) (f) (c00 && c01) || (c10 && c11)

(g) (c00 ? c01 : c02) ? c1 : c2 (h) (c00 ? c01 : c02) ? (c10 ? c11 : c12): (c20 ? c21 : c22) (i) (c0 ? c1 : c2) && (c3 ? c4 : c5) (j) (c0 ? c1 : c2) || (c3 ? c4 : c5)

(k) ((c00 ? c01 : c02) ? i1 : i2) == val (l) ((c00 ? c01 : c02) ? (c1 ? i1 : i2) : (c2 ? i3 : i4)) == val (m) (c0 ? i1 : i2) == val1 && (c1 ? i3 : i4) == val2 (n) (c0 ? i1 :i2) == val1 || (c1? i3 :i4) == val2

Figure 2: Constructing the boolean expressions

LEMMA 1. *The language of non-negative conditional expressions is defined by the BNF*

$$c ::= a \mid c_1 \;\&\&\; c_2 \mid c_1 \;\|\; c_2 \mid c_0?c_1 : c_2 \mid (c_0?i1 : i2) == val$$

$$\mid (c_0?o1 : o2).boolfunc()$$

Here a is a boolean atom, such as true, false, $x < y$, $x > y$, $x == y$, etc. $i1$, $i2$ and val are primitive variables of type int, float, etc. $o1$, $o2$ are object variables and $boolfunc$ is a function that returns a boolean value such as .equals().

For this language of non-negative conditional expressions, every CFG generated for the program segment `if c then` $S_{\mathtt{true}}$ `else` $S_{\mathtt{false}}$ *may be transformed into one that preserves the property that all incoming edges to any node in the CFG are of the same color.*

The proof of the lemma proceeds by induction on the structure of non-negative conditional expressions and is shown diagrammatically in the figures shown in Figure 2.

However as shown in figures 1(c) and 1(d), the same color property may be violated in the presence of negation. The negation of a condition involves exchanging the colors of the out-going edges. In the presence of negations in the expressions we may use the following identities on boolean expressions to push the negation inwards to the individual atoms.

$$\begin{array}{lclclcl} !!c' & = & c' & , & !(c_0?c_1 : c_2) & = & !c_0?!c_2 :!c_1 \\ !(c_1\&\&c_2) & = & !c_1\|!c_2 & , & !(c_1\|c_2) & = & !c_1\&\&!c_2 \end{array}$$

The presence of negative literals in a condition is then solved by noting that boolean atoms in Java bytecode occur as complementary pairs.

$$a \;::=\; x == y \mid x != y \mid x < y \mid x >= y \mid x > y \mid x <= y$$

We use the following identities to generate non-negative atoms wherever necessary.

$$\begin{array}{lclclcl} !(x == y) & = & (x != y) & , & !(x != y) & = & (x == y) \\ !(x < y) & = & (x >= y) & , & !(x >= y) & = & (x < y) \\ !(x > y) & = & (x <= y) & , & !(x <= y) & = & (x > y) \end{array}$$

Hence for example if some non-root node $S1$ in the subgraph has all green incoming edges and (say) a single red edge from a condition c in the subgraph, then we may switch the colors of the outgoing edges of c by negating the condition (called *twist*) in algorithm 2.

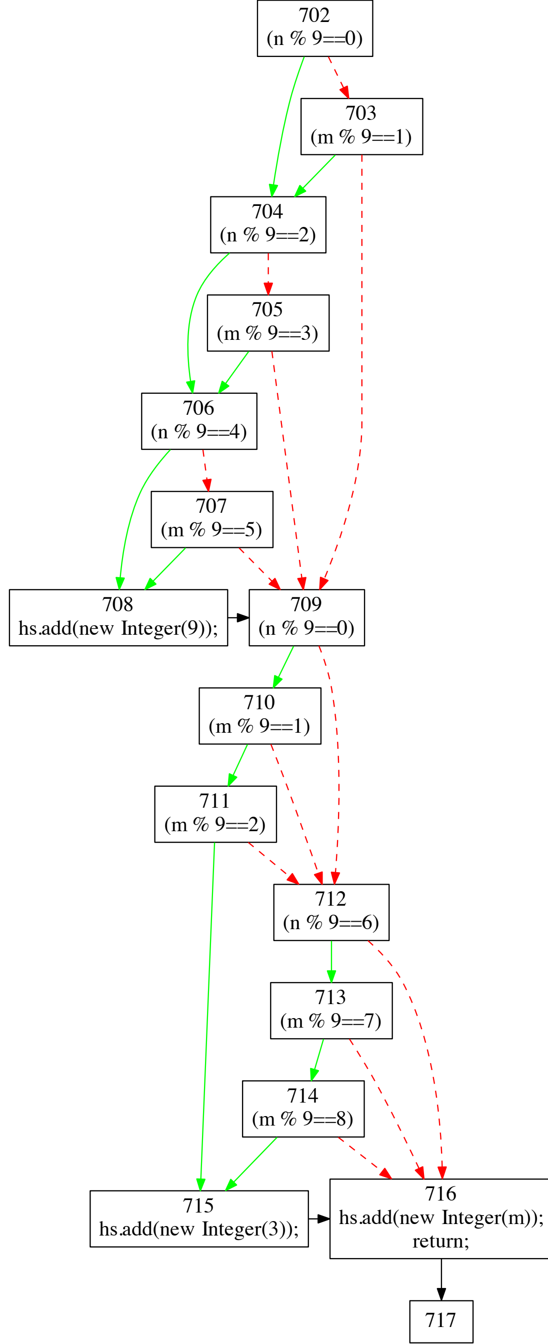
Since the the number of such nodes in the subgraph is finite, at worst, this process will end after negating every atom in the subgraph.

The algorithm. As explained in Section 1, the actual edges generated by the bytecode analysis engine may not obey this theorem. Hence, our first task is to "twist" the edges around such that in the final graph this rule is strictly applicable. The algorithm is given in Algorithm 2.

After the edges have been sorted out, we need to walk the graph recursively, detecting the four basic patterns shown in Figure 2 (a)-(d). Recognizing the patterns in Figure 2(c),

---

if-then-else statement with unstructured jumps into it (as may well happen while compiling certain FORTRAN or BASIC programs).




```
void andor(Vector hs, int m, int n) {
  if ( (n % 9 == 0 || m % 9 == 1) &&
       (n % 9 == 2 || m % 9 == 3) &&
       (n % 9 == 4 || m % 9 == 5) ) {
    hs.add(new Integer(9));
  }
  if ( (n % 9 == 0 && m % 9 == 1 && m % 9 == 2) ||
       (n % 9 == 6 && m % 9 == 7 && m % 9 == 8) ) {
    hs.add(new Integer(3));
  }
  hs.add(new Integer(m));
}
```

Figure 3: Boolean expressions

Algorithm 2 Setting up the green and red edges

> **for all** BasicBlock *bb* **do**
> $edType \leftarrow EDGEFALSE$
> $tfpred \leftarrow$ Set of true/false predecessors of *bb*
>   **if** $tfpred.size() > 1$ **then**
>   **for all** $pred \in tfpred$ **do**
>   **if** $marked(pred)$ **then**
> $edType \leftarrow edgetype(pred, bb)$
>   break
>   **end if**
>   **end for**
>   **for all** $pred \in tfpred$ **do**
> $edge \leftarrow edgetype(pred, bb)$
>   **if** $edge \neq edType$ **then**
> $twist(pred, bb)$
> $mark(pred)$
>   **end if**
>   **end for**
>   **end if**
> **end for**

(d) and(e) is fairly straightforward. For example for the pattern in Figure 2(c), all we need to do is to walk the graph, looking for a node C_0 such that C_0 has a `green` successor C_1 and C_0 has a `red` successor C_2 such that the `green` successors of both C_1 and C_2 are the same node S_T and such that the `red` successors of both C_1 and C_2 are also the same node S_F. Then C_0, C_1 and C_2 can be reduced to a single node with the expression $C_0?C_1 : C_2$ such that the predecessors of the reduced node are the same as the predecessors of C_0 and the `green` successor of the reduced node is S_T and the `red` successor is S_F.

Recognizing the patterns in Figure 2(d) is also straightforward. However, these are the only two cases where we do not make a judgement call based merely on the structure of the graph, but need to inspect the code as well. We need to locate a node C_0 such that C_0 has a `green` successor C_1 and C_0 has a `red` successor C_2. Both C_1 and C_2 have exactly one `black` successor each and both the successors are the same node, say C_3. Now we need to check the code in C_1 and C_2 and check if the same variable `var` is being assigned in both the nodes. Finally we need to check if C_4 also contains a boolean expression based on `var`.

However the patterns in Figure 2(a) and (b) are not so easy. In order to recognize them, we state (without proof) the following theorem

THEOREM 2. *At the root node N_{root} of an expression consisting of* $||$*s and* &&*s only let N_G be the destination of the* `green` *edge and let N_R be the destination of the* `red` *edge. Either N_G is the destination of multiple* `green` *edges or N_R is the destination of multiple* `red` *edges, but not both.*

Let N_G be the destination of n_g edges. Then starting from N_{root} it is possible to traverse n_g `red` *edges to reach N_Z. The* `red` *predecessor of N_Z that has the highest topological order number is the anchor N_A of the boolean expression and all nodes from N_{root} to N_A will be replaced by a single node whose incoming edges are the same as N_{root} and the outgoing edges are the same as N_A and the conditional is the equivalent of the boolean subgraph.*

And symmetrically for N_R,

EXAMPLE 1. *Consider the graph shown in Figure 3. Let*

N_{702} be the root node N_{root}. N_{704} is N_G and N_{703} is N_R. Here n_g is 2 and N_Z is N_{709} and N_A is N_{707}, Similarly for the root at N_{709}, N_{710} is N_G and N_{712} is N_R. Here n_r is 3 and N_Z is N_{716} and N_A is N_{714},

Once the scope of the expression has been identified, we need to walk the nodes in the expression in topological order. Every time we traverse a `green` edge we add an && to the expression and every time we traverse a `red` edge we add an || to the expression. We also output braces in the expression, where the spacing of the braces is dictated by the edges that are not traversed.

EXAMPLE 2. *Starting with the root at N_{709}, we walk the nodes in topological order and generate the expression for*

$(N_{709}$ && N_{710} && N_{711}) || (N_{712} && N_{713} && $N_{714})$

Observe that we do not traverse the `green` *edges from N_{709} to N_{712} or from N_{710} to N_{712} and it is this clustering of edges that determines where the braces go.*

Before closing this section, we would like to walk you through one more example with mixed boolean and ternary clauses. Consider the program shown in Figure 4. This contains almost the same expression we had introduced in the abstract. The CFG for the program is given in Figure 4 (a). In the first step we recognize the ternary subexpression $((c20?i21 * 2/3 + 4 : i22 * 2/3 + 4) == val)$. In the process, we remove the nodes N_{3204}, N_{3205} and N_{3206}. We add the subexpression to N_{3207} and N_{3207} keeps its outgoing edges, loses its incoming edges and gets a new `green-edge` from N_{3203} to N_{3207}. The resulting graph is shown in Figure 4 (b).

Next, we check the expression that starts at N_{3202}. Here N_{ROOT} is N_{3202}, n_R is 2, N_Z is N_{3207} and N_A is N_{3203}. N_R and N_G are N_{3211} and N_{3210} respectively. Then starting at N_{ROOT} we walk the graph in topological order until we reach N_A, generating the expression ($!c01$&&$!c02$). This expression is inserted into N_{ROOT} and the rest of the nodes that are traversed are deleted. Finally the outgoing edges from N_{ROOT} are deleted and replaced by edges to N_R and to N_G. The resulting graph is shown in Figure 4 (c).

In the next step, similarly, the expression rooted at N_{3208} and anchored at N_{3209} is reduced to ($c11$&&$c12$) (shown in Figure 4 (d)). Now the base ternary expression is clearly visible and it is finally reduced to the expression shown in Figure 4 (e).

Discussion. We always process a ternary expression first, that is, if it is well-formed and visible. Then we process `AND` / `OR` chains. This is because a ternary is a “stand alone” atom that may be a part of an `AND` / `OR` chain.

The expression generated is not quite identical to the original input expression although it is functionally equivalent. This is because WALA has reversed the first conditional. While, in general, it is not possible to guarantee regeneration of the original expression, we do guarantee to generate human readable code that does not contain explicit `goto`s and that does not contain labeled break statements either. The essential structure of the input expression and the generated code remains the same.

In some rare cases as at node N_{3204} in Figure 4 (a), the `green edge` is rendered to the right of the `red edge`. This is because all the programs are generated programmatically and then rendered using the `dot` program[3] from Graphviz [5]. The `dot` software sometimes gets confused and reverses the edges. However, the order of nodes remains true to the scheme and hence the node at the end of the `green edge`, N_{3205}, is numbered before the node at the end of the `red edge`, N_{3206}.

Table 1: Subject programs used in the empirical studies.

Subject	Classes	Methods	Bytecode instructions
ANTLR	507	2582	103797
XERCES	51	341	14585
DAO	3	30	930
APP A	29	195	2588
APP B	243	2134	36997
APP C	94	749	39769

4. EXPERIMENTAL RESULTS

In this section we give empirical evaluation of JINXGO[4] the tool that incorporates the algorithms presented in this paper. We developed our tool and ran these experiments on a MacBook Pro running OS X Yosemite, Version 10.10.4 on an Intel Processor, 2.5 Ghz, Intel Core i7, with 8GB of RAM.

We ran the tool on several open source and proprietary software packages, the details of which are given in Table 1.

To evaluate our approach, we conducted empirical studies using three open-source projects and three commercial products (referred to as `APP A`, `APP B`, and `APP C`)[5]. Table 1 lists the subject programs, along with the number of classes, methods, and bytecode instructions in each subject.

The decompiling boolean expressions algorithm is implemented in our tool JINXGO, using the WALA analysis infrastructure.[6] WALA includes a Java bytecode analyzer that takes Java bytecode and performs interprocedural control-flow analysis. In general this part of the analysis is performed after the program has been sliced. However, for the purpose of evaluating the algorithm, we have run the analysis on the whole program. The analysis is run immediately after the pointer analysis, which in turn runs immediately after WALA completes parsing the input bytecode, generating the control flow graphs for each method and building the call graph. The call graph is required to perform the context and path sensitive pointer analysis, which is a bit of an overkill for the decompilation process, but since it is required for the slicing algorithm, we leave it in anyway.

We conducted empirical studies to evaluate: (1) the efficiency, and (2) the efficacy of the analysis.

4.1 Efficiency

Goals and method. To evaluate efficiency, we collected data about the total analysis time, which consists of the

[3] `i=$i%.dot dot -Tpdf -o pdfs/$i.pdf $i.dot`

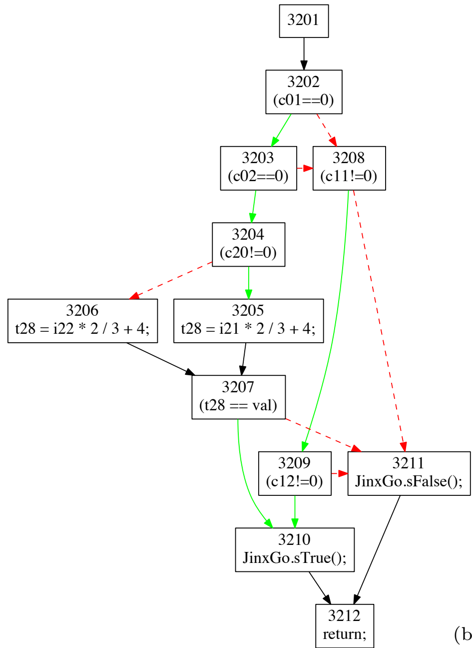
[4] As the original motivation for this tool was to detect memory leaks in Java programs, we call our tool JINXGO which is a parody on Ginkgo Biloba which is a tree whose leaves are used in treating Alzheimer's disease, the human memory disorder.

[5] IBM confidential

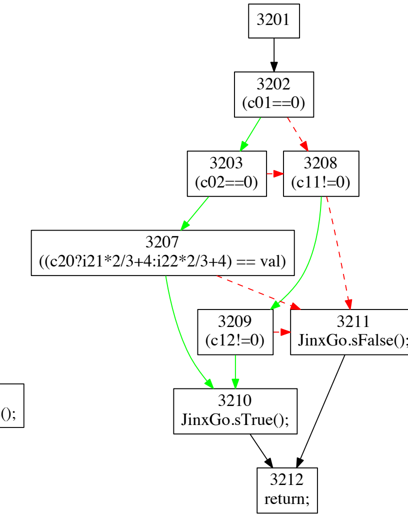
[6] http://wala.sourceforge.net

The original input program

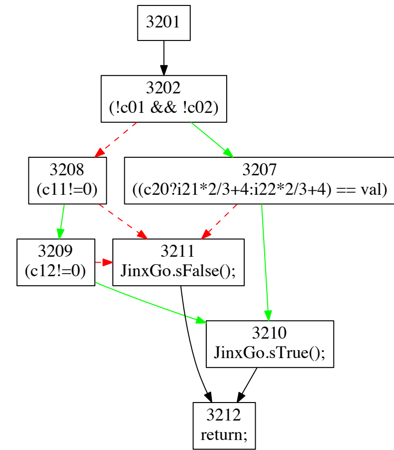
```
if ( (c01 || c02) ? (c11 && c12) : (c20?i21*2/3+4:i22*2/3+4) == val ) {
  sTrue();
}
else {
  sFalse();
}
```



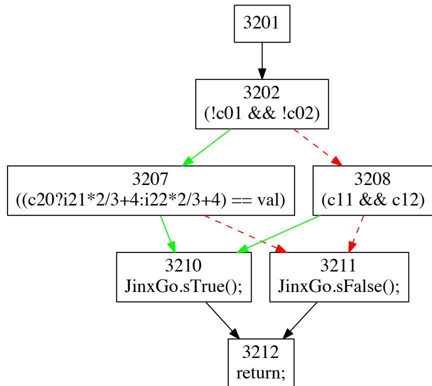
(a) The CFG generated by WALA for the given program.



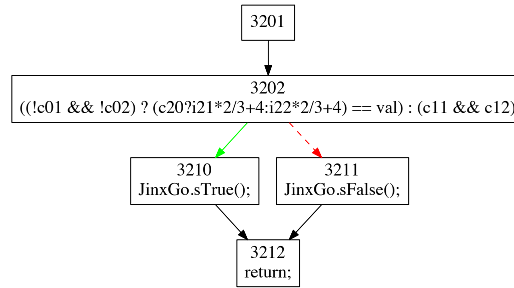
(b) The CFG after replacing the ternary expression by a single node.



(c) The CFG after replacing the first boolean expression also by a single node.



(d) The CFG after replacing all the top level boolean expressions. The second layer ternary expression is now exposed.



(e) The original program regenerated, albeit with some structural (but equivalent) changes.

Figure 4: Decompiling an expression with Simple and Ternary Boolean Sub-expressions

Table 2: **Time for analysis in seconds.**

Subject	Parsing Building	Pointer Analysis	Decompiling expressions	Total time
Antlr	25	24	1	51
Xerces	8	5	0	14
Dao	2	0	0	2
App A	7	0	0	8
App B	11	2	0	15
App C	6	4	1	11

Table 3: Number of AND and OR clauses.

Subject	exactly 2	exactly 3	>3	Total expressions	max size
Antlr	255	64	48	367	19
Xerces	98	24	25	147	10
Dao	10	0	6	16	7
App A	12	0	1	13	7
App B	233	50	33	316	13
App C	86	16	8	110	8

time required to perform the preliminary analysis and the time required to perform the decompilation.

Results and analysis. Table 2 shows the execution time (in seconds) for preliminary analysis and the time to decompile.

Discussion. The data demonstrates the efficiency of the analysis: our analysis typically runs in less than a second. The maximum time taken is by the WALA infrastructure to parse the bytecode and build the call graph, followed by the pointer analysis.

4.2 Efficacy

Goals and method. To evaluate the efficacy of our algorithms, we gathered numbers for the total number of expressions found in a package. Of these we counted the number of expressions that had exactly 2 subexpressions, exactly 3, more than 3 and also determined the size of the largest expression that contained only AND and OR clauses.

Next we counted the number of expressions that had exactly 1 subexpression, exactly 2, more than 2 and also checked the size of the largest expression that contained only ternary subexpressions.

Finally we counted the number of expressions that had exactly 2 sub-expressions, exactly 3, more than 3 and also checked the size of the largest expression that contained a mix of AND and OR clauses as well as ternary sub-expressions.

Results and analysis. Table 2 shows the execution time (in seconds) for preliminary analysis and the time to decompile.

Discussion. Tables 3 – 5 give the data relevant to this discussion. As can be seen, all the packages generate several conditionals with AND and OR clauses and some of them generate truly very long expressions with as many as 19 clauses.

However, very few packages generate ternary clauses and none of them generate expressions that have more than one ternary subexpression. The package Dao may appear to be an exception, but that is because it is our own package that we used for testing our algorithms. We were delighted to see that it handled whatever we threw at it.

Table 4: Number of Ternary clauses.

Subject	exactly 1	exactly 2	>2	Total expressions	max size
Antlr	14	-	-	14	1
Xerces	-	-	-	-	1
Dao	23	9	19	51	19
App A	2	-	-	2	1
App B	10	-	-	10	1
App C	2	-	-	2	1

Table 5: Number of mixed AND and OR and ternary clauses detected "C" correctly, in a "Messed"up way / "I" Incorrectly by each tool.

Tool Name	exactly 2		≥3		Total	
	C	M/I	C	M/I	C	M/I
JinxGo	2	0/0	4	0/0	7	0/0
Jode	0	2/0	0	4/0	0	6/0
JReversePro	0	2/0	4	4/0	7	7/0
Procyon	2	0/0	0	4/0	2	4/0
FernFlower	2	0/0	0	4/0	2	4/0
CFR	2	0/0	2	0/2	2	2/2
Soot	0	2/0	0	4/0	0	6/0

None of the packages generated a mix of ternary and AND/OR expressions. So, in table 5 we give only the results for our experimental package, Dao, with a comparison of all the tools that we have tested. Essentially all the tools, other than CFR, generate correct code albeit "messed up" (where by "messed up" we mean code that involves labeled breaks). CFR actually generates correct code for complex expressions like `(m % 9 == 0 && ((n = m % 9) == 1 ? ((n = m % 8) == 2 ? n + m < 4 : n - m < 4) : n - m < 3))` but generates incorrect code for similar expressions where AND is replaced by OR. WE do not know whether it is a bug in the algorithm or just an implementation bug.

5. RELATED WORK

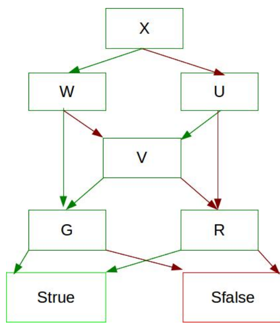
Venkatesh [16] classifies static slicing algorithms as "executable" and "closure" slices. Closure slices contain the set of statements that are related to the variable of interest through a closure of dependencies and are not necessarily either syntactically correct or executable programs, *i.e.,* programs which on execution preserve the behavior of the original program. Weiser[17]'s algorithm produces executable slices. However, his algorithm does not produce precise slices for programs with procedures since it fails to account for the calling context of procedures. Horwitz *et al* [8] were the first to address the issue of calling contexts and gave a closure based context-sensitive slicing algorithm for slicing programs with procedures. Papers on generating semantically correct slices for sequential programs include [6, 18, 3].

In this paper we have presented an algorithm for generating code for boolean expressions for "human readable" executable slices for sequential Java programs.

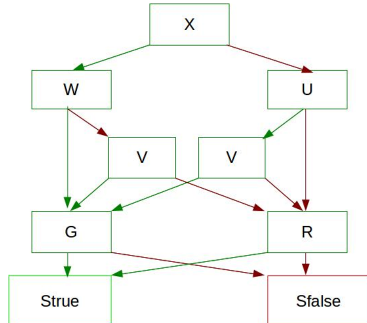
Although, there are a number of open-source Java decompilers, only a few of them provide a formal literature. Miecznikowski and Hendren [11] were the first to formally present an algorithm for Decompilers. They present a decompiler DAVA (present as an option in Soot [15]) to regenerate the input program. DAVA follows a six stage decompilation process to emit the source code from the input

The original input program

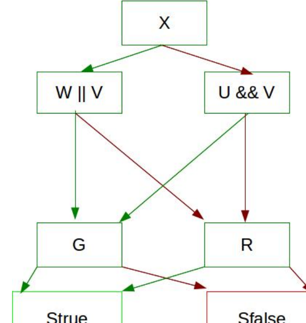
```
if((X?(W||V):(U&&V))?G:R) {
      sTrue();
    }
    else {
      sFalse();
    }
```



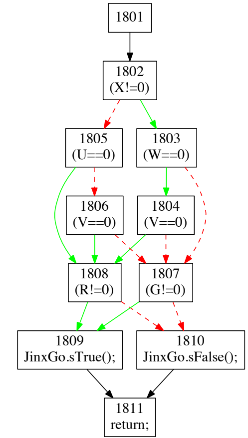
(a) A CFG with an untwistable DAG.



(b) Equivalent CFG after duplicating a node.



(c) The simplified CFG.



(d) Generated graph equivalent to the graph in Figure 5(b)

Figure 5: Managing untwistable DAGs

bytecode. Their implementation utilizes three representations – Grimp, CFG and SET – to facilitate decompilation.

Naeem and Hendren [12] present a user friendly decompiler. The authors modify the DAVA decompiler, to generate programmer-friendly code. The authors present an algorithm that applies a series of transformations such as And Aggregation, Or Aggregation, Useless Label Remover, Loop Strengthening and Condition Simplification, to generate a simplified code. Yet, as can be seen from the actual output generated, the code is not really user friendly.

Our decompiler requires only two intermediate representations: the topological ordering of graphs and the control flow graph. We present a discussion on the re-generation of complex expressions.

6. THE UNTWISTABLE DAG

The CFG shown in Figure 5(a) is an example of a CFG (a basic "untwistable" DAG) which violates theorem 1. On the other hand such a CFG would never be generated automatically on any expression in the language of conditional expressions as lemma 1 shows. It is a manually drawn CFG and the expression has been derived from the graph. We ran the expression through our software to validate the expression and generated the plot shown in Figure 5(d) which is equivalent to the graph in Figure 5(b). A byte-code optimizer could, however transform a DAsG of the form Figure 5(b) into one of the form shown in Figure 5(a).

The way to handle these untwistable DAGs is to duplicate the node (or nodes) that remain untwistable such that each copy of the node has exactly one color of incoming edges. Then the algorithm continues as before. For example, in Figure 5(b) we have duplicated the node V because it has both an incoming `green edge` and an incoming `red edge` and it is not possible to twist the conditions so as to make them both either `green` or `red`.

Once the node has been split it is easy to see that W and V combine to generate $W||V$ and that U and V combine to generate $U\&\&V$. This exposes a classic ternary expression (shown in Figure 5(c)) which generates $X?(W||V) : (U\&\&V)$ which in turn exposes yet another classic ternary expression (not shown) which generates the final expression $(X?(W||V) : (U\&\&V))?G : R$.

We believe that this kind of DAG may be generated by some extremely aggressive common sub-expression elimination algorithm. Which would explain why the common sub-expression V has been combined into a single node.

7. CONCLUSION

The paper arose out of our attempt at program slicing to produce executable slices. In general, the problem of obtaining executable slices is quite hard. As is usual in any large programming language, it is necessary to begin this process from some low-level architecture-independent representation of programs. We have chosen Java bytecode as the starting point, from which we reconstruct a Java program slice. The resulting program therefore, will not correspond exactly (at the expression-level in a syntactic sense) to the original source. However we do believe that the effects on the classes of interest will help in debugging the original source code. Debugging is aided considerably if the executable code produced is free from `goto` and arbitrary jumps.

However even for structured programs most code generation techniques use `goto` in the byte-code, especially when the evaluation of complex boolean conditions is involved.

The resulting byte-code is more often than not unreadable and hence makes debugging very hard. This paper is the result of trying to produce human-readable ('`goto`-less') source code from byte-code.

We have a working implementation of the techniques we have presented here, though it is not ready yet for commercial use. We have tested it on several samples of code.

Our experience in this project has given us some insights into obfuscation of code. Obfuscation of the byte-code can make it very hard for a decompiler to generate source code that is readable or comprehensible. In such cases, the only way to get around obfuscation may be by running a dead-code elimination routine on the bytecode before decompiling it.

The obvious next step in this project is to study techniques for producing executable slices of concurrent Java threads. It has been well recognized that reasoning about and debugging concurrent programs is far more difficult and it would be very useful to have slicing techniques for concurrent programs which produce executable code. Hitherto attempts at slicing concurrent programs [10] have not yielded executable versions.

8. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Benf. Cfr - another java decompiler. http://www.benf.org/other/cfr/.

[3] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, January 1995.

[4] I. Corporation. Wala: The T. J. Watson Libraries for Analysis. http://wala.sourceforge.net.

[5] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000. URL http://www.graphviz.org.

[6] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, 2003.

[7] J. Hoenicke. Jode. http://jode.sourceforge.net/.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Transactions on Programming Languages and Systems*, 12:26–60, Jan 1990.

[9] K. K. Jreversepro - java decompiler / disassembler. http://jreversepro.blogspot.in/.

[10] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering*, 2003.

[11] J. Miecznikowski and L. J. Hendren. Decompiling java using staged encapsulation. In *WCRE*, pages 368–374, 2001.

[12] N. A. Naeem and L. J. Hendren. Programmer-friendly decompiled java. In *ICPC*, pages 327–336, 2006.

[13] R. Shevchenko. Fernflower java decompiler. https://github.com/fesh0r/fernflower.

[14] M. Strobel. Procyon / java decompiler. https://bitbucket.org/mstrobel/procyon/wiki/Java Decompiler.

[15] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[16] R. Venkatesh. The semantic approach to program slicing. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1991.

[17] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.

[18] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology*, 1(3):310–354, July 1992.