Fast Symbolic Evaluation of C/C++ Preprocessing Using Conditional Values

Mario Latendresse
NGIT/FNMOC
7 Grace Hopper, Monterey, CA, USA.
E-mail: latendre@metnet.navy.mil

Abstract

C/C++ code relying on preprocessing can be quite complex to analyze. This is often due to free preprocessing variables set at compile time. In general, preprocessing selectively compile parts of the source code based on the values of preprocessing variables which may be free. In this case, the relations between these parts can only be represented by conditional expressions using the free variables. Traditional symbolic evaluation can be used to infer these expressions, but its best case time complexity is exponential. We present a new approach for symbolic evaluation that can efficiently compute these conditions by binding variables to conditional values and avoiding the path feasibility analysis of traditional symbolic evaluation. It infers the exact conditional expressions for which the lines of code are compiled and the (conditional) values of preprocessing variables at each point of the source code. Our prototype shows the approach as practical and scaleable to large C/C++ software.

```
1.  #if Y==1
2.   #define A 2
3.  #endif
4.
5.  #if Y==2
6.   #define B 4
7.  #endif
8.
9.  #if Y==3
10.  #define C 8
11. #endif
12.
13. #if Y==4
14.  #define D 16
15. #endif
16.
17. #if defined(D)
18.  int x;
19. #else
20.  char x;
21. #endif
```

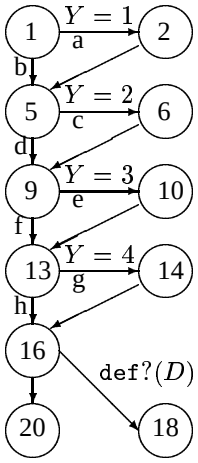

Figure 1. On the left, a C source code, on the right, its CFG.

1. Introduction

In C/C++, preprocessing is done by `cpp`, prior to compilation itself. Automatic C/C++ code analysis and maintenance traditionally assume that the preprocessing phase has been done. In practice, free preprocessing variables, that is variables set only at preprocessing time outside the source code, must be set at some specific values to call `cpp`. This is unsatisfactory since a large number of combinations of values are possible (for example see [3, 11]).

A more acceptable approach is to compute, for each line of code, the conditional expression that determines its reachability (compilation), based on the free variables. Moreover, the values of each preprocessing (non-free) variables should be determined based on similar conditional expressions. With such fundamental information, more complex analyses can be done.

For example, if the condition to reach a line is contradictory, this line can never be compiled, and therefore points to an erroneous design.

In Fig. 1, the lines beginning by '#' are preprocessing directives (spaces may precede '#'). In this case, the directives are `#define` and `#if`, the latter is composed of a then- and/or else-block formed by the `#else` and `#endif` lines. A define directive binds a preprocessing variable, thereafter simply called a variable, with a list of tokens. If the definition is parameterized, it is a macro. If the list of tokens is empty, as in `#define X`, we denote by $\top$ the value of X. Such variables are bound. If a variable is not defined, that is unbound, we denote its value by $\bot$. A programmer can explicitly unbound a variable by using the `#undef` directive. All bindings are global, that is, once an identifier is defined its value is visible by all preprocessing directives. On the right side of Fig. 1 is the control flow graph (CFG) of the source code: A node embodies a block of lines of the source code; an arc a branching decision. For two out

arcs of a node, one is labeled by the condition to take it, the other one is taken when the condition is `false`. The predicate `def?`(x) is `true` if and only if the variable x is defined, that is, if its value is not $\perp$.

In symbolic evaluation, the initial values of variables are unknown. We use the symbol x_I to represent the value of variable x before preprocessing. The value of x_I is one of the preprocessing values, that is $\perp$, $\top$, or a list of (parameterized) tokens.

Conditional compilation occurs when if-directives are used. They adapt parts of code to hardware, operating systems, software version, etc. For example, line 18 is compiled only if D is defined. Line 14 is reached if the initial value of Y is 4. The series of if-directives determine a type for the C variable `x`.

The variables Y and D are not explicitly bound before all their references. They are *free variables*. They can be bound at compile time using some other means — for example, on the command line calling the compiler. Note that such variables are not considered unbound. Actually, their values is symbolically represented as Y_I and D_I, respectively.

What is the condition to compile line 18, that is for `x` to be of type `int`? The answer is easily seen: if D is defined prior to preprocessing or if Y_I is 4 then the type of `x` is `int` otherwise it is `char`.

A programmer maintaining such code may be trying to answer several questions regarding the conditional compilation. For example, are lines 10 and 18 mutually exclusive under all values of the free variables? (They are not.) What are the possible values of D at line 16 and for which conditions does it have these values? Are there lines which are never compiled or reached under all possible values of the free variables? Etc.

To answer these questions it is necessary to find, for each line, the conditions for which it is compiled and the possible values of variables. Therefore, these are the basic problems addressed in this paper:

1. For each line of source code, what is the condition for which this line is compiled or reached?
2. For each line referring to a preprocessing variable, what are its possible values and under which conditions do they have these values?

For the code of Fig. 1, this can be done manually, but for large systems of thousands of lines, this is impractical.

The traditional symbolic evaluation, pioneered by [6] and used in [4] for C/C++ preprocessing, cannot be used in practice to solve these problems, since its best case complexity is exponential on the number of conditional branches.

Traditional symbolic evaluation traverses all paths of the CFG. In case of Fig. 1, they are sixteen paths formed by the first four if-directives. These can be labeled by the conditions that are considered `true` or `false`. For example, (a, d, f, g) is a possible path. Yet it is contradictory, since Y cannot be equal to 1 and 4 at the same time, in other words this is not a feasible path. Only five are feasible (e.g. (a, d, f, h)). Traditional symbolic evaluation, as used in [4], consider these sixteen paths, and combines disjunctively the sixteen conditions to form the full condition to compile `int x;`. The resulting condition is, relative to the number of if-directives, exponential in size. The computational complexity of this technique is not only in $O(2^n)$ but in $\Theta(2^n)$, where n is the number of sequential if-directives. That is, we cannot even expect an average computational complexity lower than exponential.

If three files, each having six if-directives, are included at the beginning of Fig. 1, using the include directive, the number of paths to reach `int x;` would increase to $2^{3\times6+4}$. That is, over four millions paths would have to be considered. The size of the disjunctive Boolean expression would be very large, despite the fact that all these included if-directives are probably irrelevant for the condition to reach line 18.

In this paper we present a new symbolic evaluation approach that can, in practice, efficiently solve the two main problems. This approach is based on *conditional values*.

It is also necessary to determine the satisfiability of conditions, since iteration is possible with the include directive. Although the general problem is NP-complete, we use appropriate simplification rules for conditional values to bring practical efficiency.

This paper is organized as follows. Section 2 introduces conditional values, the fundamental technique of our approach. Simplifications of conditions with conditional values are presented in Section 3. In Section 4 our general symbolic evaluation algorithm is presented. Section 5 presents two concrete examples to illustrate our symbolic evaluation using conditional values. Section 6 addresses the problem of iterations. Section 7 discusses syntactical cases for which conditional macro expansion is complex. Section 8 presents experimental results of our prototype. Related works are presented in 9. We conclude in section 10.

2. Conditional values

Our approach uses a new symbolic representation called *conditional value* or *c-value* denoted $c \rightarrow e_1 \diamond e_2$ where c is a conditional expression and the expressions e_1 and e_2 are c-values or base values. It is interpreted as: if c is `true` then its value is e_1 otherwise it is e_2. Since c-values may be nested, they can represent all base values a variable has at a particular line of the source code.

For example, in Fig. 2, the value of W at line 6 is $Y_I = 1 \rightarrow 2 \diamond 3$. It indeed represents the value of W af-

```
1.  #if Y == 1
2.    #define W 2
3.  #else
4.    #define W 3
5.  #endif
6.
7.  #ifdef Y
8.    #define U  Y
9.  #else
10.   #define U  W
11. #endif
```

Figure 2. Two c-values are generated by this code; $Y_I = 1 \rightarrow 2 \diamond 3$ for W and $\texttt{def?}(Y_I) \rightarrow Y_I \diamond (Y_I = 1 \rightarrow 2 \diamond 3)$ for U.

ter the if-directive since the value of W is 2 if the initial value of Y is 1, otherwise it is 3. Note that the Boolean expression uses the symbol Y_I and not the symbol Y, since the Boolean expression refers to the initial value of the variable. After the second if-directive, at line 12, the c-value of U is $\texttt{def?}(Y_I) \rightarrow Y_I \diamond (Y_I = 1 \rightarrow 2 \diamond 3)$. The c-value of U has a nested c-value, namely the c-value of W. Indeed, in general c-values may be nested.

The notion of c-values avoids the combinatorial analysis of paths. This is our fundamental means to avoid the major pitfall of traditional symbolic evaluation.

With the analysis of all paths that may reach a line, any symbolic evaluation would require an exponential time, relative to the number of if-directives. On the other hand, by using c-values this analysis is no longer required. The next section presents simplification rules for c-values.

3. Simplification of conditional values

In this section we look at simplification rules on c-values. They are useful when they help determine the status (e.g. satisfiable, tautology, contradiction) of Boolean expressions with c-values.

We also apply common simplification rules on negation and Boolean connectives $\wedge$ and $\vee$ as well as transformations to reduce the size of a Boolean expression through Boolean algebra. For example, $c \vee \texttt{false} \equiv c$.

Conditional values and the predicate `def?` provide new opportunities for simplifications. The fundamental rules used, expressed as equivalences, are given in Fig. 3. Some more rules, or equivalences, which are derived from these fundamentals equivalences are presented in Fig. 4.

Fundamental rule 1 is used only if e_1 and e_2 are of type Boolean; some common particular cases are derived to produce rules 7 to 10 of Fig. 4.

Fundamental rules 2 to 5 are specifically for the predicate `def?`. Rules 3 and 4 simply express the meanings of defined ($\top$) and undefined ($\bot$). Rule 5 represents a very common

1. $c \rightarrow e_1 \diamond e_2 \equiv c \wedge e_1 \vee \neg c \wedge e_2$
   where e_1 and e_2 are Boolean expressions.
2. $\texttt{def?}(c \rightarrow e_1 \diamond e_2) \equiv c \rightarrow \texttt{def?}(e_1) \diamond \texttt{def?}(e_2)$
3. $\texttt{def?}(\bot) \equiv \texttt{false}$
4. $\texttt{def?}(\top) \equiv \texttt{true}$
5. $\texttt{def?}(tokens) \equiv \texttt{true}$
6. $(c \rightarrow e_1 \diamond e_2) \quad t \quad e_3 \equiv c \rightarrow (e_1\ t\ e_3) \diamond (e_2\ t\ e_3)$
   where t is any token.
7. $c_1 \rightarrow (c_2 \rightarrow e_1 \diamond e_2) \diamond e_3 \equiv (c_1 \wedge c_2) \rightarrow e_1 \diamond (c_1 \rightarrow e_2 \diamond e_3)$
8. $c_1 \rightarrow e_1 \diamond (c_2 \rightarrow e_2 \diamond e_3) \equiv (\neg c_1 \wedge c_2) \rightarrow e_2 \diamond (c_1 \rightarrow e_1 \diamond e_3)$
9. $c \rightarrow e \diamond e \equiv e$

Figure 3. Fundamental equivalences to simplify c-values.

case: a variable that has a specific value, that is a sequence of tokens, is defined. For example, $\texttt{def?}(Y_I > 3 \rightarrow 4 \diamond \top)$ is equivalent to $Y_I > 3 \rightarrow \texttt{def?}(4) \diamond \texttt{def?}(\top)$ according to rule 2; it can be further simplified to $Y_I > 3 \rightarrow \texttt{true} \diamond \texttt{true}$ by rules 4 and 5; and finally to `true` by the derived rule 7.

Rule 6 is very general but used mostly when t is a relational or arithmetic operator. For example, applying it to $(Y_I = 1 \rightarrow 2 \diamond 3) > 1$ gives $(Y_I = 1) \rightarrow (2 > 1) \diamond (3 > 1)$ which is equivalent to $(Y_I = 1) \rightarrow \texttt{true} \diamond \texttt{true}$ which is `true` by the derived rule 7.

Rules 7, 8 and 9 are often used in conjunction to simplify c-values of the form $c_1 \rightarrow (c_2 \rightarrow e_1 \diamond e_2) \diamond e_3$ or $c_1 \rightarrow e_1 \diamond (c_2 \rightarrow e_2 \diamond e_3)$ where a pair e_i are equal. For example, the c-value $X = 2 \rightarrow (Y = 3 \rightarrow 4 \diamond 5) \diamond 5$ can be simplified to $(X = 2 \wedge Y = 3) \rightarrow 4 \diamond (Y = 3 \rightarrow 5 \diamond 5)$ by rule 7 and then to $(X = 2 \wedge Y = 3) \rightarrow 4 \diamond 5$ by rule 9. Application of these rules unfolded nested c-values.

In general it is necessary to detect when a condition is satisfiable. Otherwise, infinite iteration, due to recursive inclusion of files, would trap the symbolic evaluation. A satisfiability test is used in our general symbolic evaluation on all conditions. Such a problem is NP-complete, but the simplification rules efficiently solve most of the practical cases.

4. The symbolic evaluation algorithm

In this section we present the essential elements of our symbolic evaluation algorithm as shown in Fig. 5. The main

1. $\texttt{def?}(c \rightarrow e_1 \diamond e_2) \equiv c \wedge \texttt{def?}(e_1) \vee \neg c \wedge \texttt{def?}(e_2)$
2. $\texttt{def?}(c \rightarrow \top \diamond e) \equiv c \vee \texttt{def?}(e)$
3. $\texttt{def?}(c \rightarrow e \diamond \top) \equiv \neg c \vee \texttt{def?}(e)$
4. $\texttt{def?}(c \rightarrow \bot \diamond e) \equiv \neg c \wedge \texttt{def?}(e)$
5. $\texttt{def?}(c \rightarrow e \diamond \bot) \equiv c \wedge \texttt{def?}(e)$
6. $\texttt{def?}(\neg\texttt{def?}(x_I) \rightarrow \top \diamond x_I) \equiv \texttt{true}$
7. $c \rightarrow \texttt{true} \diamond \texttt{true} \equiv \texttt{true}$
8. $c \rightarrow \texttt{false} \diamond \texttt{false} \equiv \texttt{false}$
9. $c \rightarrow \texttt{true} \diamond \texttt{false} \equiv c$
10. $c \rightarrow \texttt{false} \diamond \texttt{true} \equiv \neg c$

Figure 4. Some derived equivalences from Fig. 3 to simplify c-values.

algorithm is from line 1 to 6, but the essential work is done by the recursive function E.

The symbolic evaluation is done on the CFG A of the source code: each node is either a preprocessing directive or it is a block of C/C++ code. Assume that each node of A is initialized with an empty list of conditions.

We could also add a list of all variable bindings for each node to fully answer the second problem mentioned in the introduction. But in practice, only some variables will be useful for further analysis and can be extracted by this algorithm as needed.

There are two important variables: c_c represents a sufficient condition for which the current node of code may be reached and a global stack S of tables. A table is a set of variable bindings, that is an association list of identifiers and values. The current table is at the top of S. The value of a variable x is the first value found by searching the tables starting from the top of S. That is, the top table of S is used first and if no binding is found for x the next table on S is used, etc. If for all tables no value is found for x, its symbolic value is x_I. We denote this search by $v(x, S)$.

At line 2, the algorithm initializes the stack S with one empty table and at line 3 calls E with the root node of A and `true` for the current condition.

After the execution of E, each node of A has a list of conditions. These lists answer the first problem formulated in section 1: The full reachability condition of a node is the oring of conditions in its list. So, for each node, the disjunction of the list of conditions forms the full condition under which this node is reached or compiled. The empty list forms the condition `false`. Also, only one table remains in

```
1.  Main
2.      Push empty table [] onto S
3.      Call E(A, true)
4.      The CFG A contains all conditions
5.      The table in S has the final variable bindings
6.  End

7.  Procedure E(n, c_c) {
8.      add c_c to condition list of node n
9.      test node n for possible infinite iteration
10.     Case node n
11.       block of C/C++ code:  nothing to do
12.       define:    add definition to top table of S
13.       if: Let c be its expanded/simplified condition
14.         if c_c ∧ c is satisfiable then {
15.               Push empty table [] onto S;
16.               Call E(n.then, c_c ∧ c)
17.               Pop top table from S and assign it to T_1
18.         } else T_1 is empty
19.         if c_c ∧ ¬c is satisfiable and n.else exists then {
20.               Push empty table [] onto S;
21.               Call E(n.else, c_c ∧ ¬c)
22.               Pop top table from S and assign it to T_2
23.         } else T_2 is empty
24.         Merge(T_1, T_2, S, c)
25.     End Case
26.     if n.next exist then Call E(n.next, c_c)
27. }

28. Procedure Merge(X, Y, S, c) {
29.     For-each variable x in X or Y
30.       Bind x with c → v_t ◇ v_f into the top table of S
31.       where  v_t is v(x, X : S)
32.              v_f is v(x, Y : S)
33. }
```

Figure 5. Our symbolic evaluation algorithm for C/C++ preprocessing.

S and it has all preprocessing variable bindings associated with their conditional values. If a preprocessing variable is unreachable, it will not be in that table.

The recursive procedure E takes two arguments, a node n and a condition c_c.

Line 8 adds the current condition c_c to the list of conditions of node n. This node may have been visited several times since an iteration may exist, so line 9 tests for a possible infinite iteration. Section 6 explains in more details this case.

There are three cases for each node of the CFG: a block of C/C++ code, a definition of a preprocessing variable (or

macro) and an if-directive. We have included the essential cases, since all the other directives (e.g. `#warning`, `#undef`, `#elif`) are either irrelevant or can be implemented using these cases.

For a block of C/C++ code, there is nothing further to do as the current condition c_c has been added to its list of conditions and no preprocessing directives have to be considered.

For a define-directive, the definition is added to the current table which is at the top of S. This algorithm accepts redefinitions but refers only to the latest definition.

For an if-directive, its condition is expanded and simplified using the rules of the previous section. By recursion, the current condition c has also been simplified. In practice, it is often the case that the simplified version becomes `true` or `false` and satisfiability becomes trivial to establish. Otherwise, a more general procedure is used. If it is satisfiable, line 15 pushes an empty table onto S. This is necessary since all directives of the block form a separate entity. At line 16, the recursive call E is made with the root of the then-block as the current node and the simplified form of $c_c \wedge c$ as the current condition. This call will use the empty table to possibly insert new bindings.

If the inverse condition is satisfiable and there is an else part, a similar recursive call is made for that block. Note that, in general, both the then- and else-block may be evaluated.

At line 24, two tables have been created, T_1 and T_2. They are merged and inserted into the top table of S by procedure Merge. Line 26 recursively iterates on the next existing node.

The merging of two tables of bindings is described in lines 28 to 33. This is where all c-values are generated. It takes two tables X and Y, a stack of tables S and a condition c. The merging operation inserts all variable bindings found in X or Y into the top table of S. The notation $Y : S$ is a stack formed by table Y on top of stack S. So, the value v_f comes from Y or one of the tables of S (this is similar for v_t but with X). The bind operation of line 30 removes any existing binding of x, if one exist. The c-value $c \rightarrow v_t \diamond v_f$ becomes the value of x in the top table of S.

Note that in E, for each table pushed onto S, a corresponding pop is done. Therefore after the initial call of E, at line 4, only one table remains in S.

5. Two examples of our symbolic algorithm

5.1. A short example

Fig. 6 shows a short example with a trace of created variable bindings. To reduce clutter, we only show the table of bindings at the top of the stack S. A couple (x, v) represents a binding between variable x and value v. A table is

0.		$[\,]_1$
1.	`#define A 1`	$[(A,1)]_1$
2.	`#if Y==2`	$[\,]_2$
3.	`  #define A 3`	$[(A,3)]_2$
4.	`#endif`	$[(A, Y_I = 2 \rightarrow 3 \diamond 1)]_1$
5.	`#if Y==4`	$[\,]_2$
6.	`  #define B 5`	$[(B,5)]_2$
7.	`  #if W==6`	$[\,]_3$
8.	`   #define C 7`	$[(C,7)]_3$
9.	`  #endif`	$[(B,5),(C, W_I = 6 \rightarrow 7 \diamond C_I)]_2$
10.	`#endif`	$[(A, Y_I = 2 \rightarrow 3 \diamond 1),$
		$(B, Y_I = 4 \rightarrow 5 \diamond B_I),$
		$(C, Y_I = 4 \rightarrow (W_I = 6 \rightarrow 7 \diamond C_I) \diamond C_I)]_1$

Figure 6. On the right is the table bindings on top of stack S.

a list of such bindings represented between square brackets $[\ldots]$. A subscript (e.g. 1) on the right of a table gives its height in the stack. For example, at line 3, the stack of tables is $[(A,1)]_1[(A,3)]_2$, with $[(A,3)]$ at the top; at line 8, the stack is $[(A, Y_I = 2 \rightarrow 3 \diamond 1)]_1[(B,5)]_2[(C,7)]_3$. At line 10, the stack is fully presented as there is only one table left at the end of the symbolic evaluation.

At line 0, an empty table is pushed on stack S as specified at line 2 of the algorithm of Fig. 5. Function E is called and since at line 1 it is the definition of variable A, this adds binding $(A, 1)$ to the top table and E is called recursively on the next node (line). At line 2 the condition c is $Y_I = 2$. It refers to the initial value of Y since it has no value in all the tables of S; and an empty table is pushed onto the stack since a then-block is entered; E is called recursively for it. A binding $(A, 3)$ is added to this new table at line 3. It returns and binds T_1 with the table $[(A, 3)]$. There is no else-block, so a merge is done with T_1 (X for Merge) and an empty T_2 (Y for Merge). The merge occurs only for this variable, it has value 3 for X and 1 for $Y : S$, creating the c-value $Y_I = 2 \rightarrow 3 \diamond 1$ for A. (Note that line 4, as any `#endif`, is not directly processed by the symbolic evaluation since it is an end marker of the then-block node in the CFG.) Lines 5, 6, 7, 8, and 9 generates similar operations. At line 10, variables B and C are merged with the table created at line 4. Note that C can be simplified to $(Y_I = 4 \wedge W_I = 6) \rightarrow 7 \diamond C_I$ using rules 7 and 9 of Fig. 3.

5.2. A longer example

Let us apply the algorithm to the code presented in Fig. 1, reproduced on the left of Fig. 7 with the table of bindings at the top of the stack and the current condition c_c.

At line 0, the current condition c_c is `true`, and the stack

Code	Current table on top of stack	Current condition
0.	$[\,]_1$	`true`
1. `#if Y==1`	$[\,]_2$	$Y_I = 1$
2. `#define A 2`	$[(A, 2)]_2$	$Y_I = 1$
3. `#endif`	$[(A, Y_I = 1 \rightarrow 2 \diamond A_I)]_1$	`true`
4.		
5. `#if Y==2`	$[\,]_2$	$Y_I = 2$
6. `#define B 4`	$[(B, 4)]_2$	$Y_I = 2$
7. `#endif`	$[(A, Y_I = 1 \rightarrow 2 \diamond A_I), (B, Y_I = 2 \rightarrow 4 \diamond B_I)]_1$	`true`
8.		
9. `#if Y==3`	$[\,]_2$	$Y_I = 3$
10. `#define C 8`	$[(C, 8)]_2$	$Y_I = 3$
11. `#endif`	$[(A, Y_I = 1 \rightarrow 2 \diamond A_I), (B, Y_I = 2 \rightarrow 4 \diamond B_I), (C, Y_I = 3 \rightarrow 8 \diamond C_I)]_1$	`true`
12.		
13. `#if Y==4`	$[\,]_2$	$Y_I = 4$
14. `#define D 16`	$[(D, 16)]_2$	$Y_I = 4$
15. `#endif`	$[(A, Y_I = 1 \rightarrow 2 \diamond A_I), (B, Y_I = 2 \rightarrow 4 \diamond B_I), (C, Y_I = 3 \rightarrow 8 \diamond C_I),$	`true`
16.	$(D, Y_I = 4 \rightarrow 16 \diamond D_I)]_1$	
17. `#if defined(D)`	$[\,]_2$	$\texttt{def}?(Y_I = 4 \rightarrow 16 \diamond D_I)$
18. `int x;`	$[\,]_2$	$\texttt{def}?(Y_I = 4 \rightarrow 16 \diamond D_I)$
19. `#else`		
20. `char x;`	$[\,]_2$	$\neg\texttt{def}?(Y_I = 4 \rightarrow 16 \diamond D_I)$
21. `#endif`	$[(A, Y_I = 1 \rightarrow 2 \diamond A_I), (B, Y_I = 2 \rightarrow 4 \diamond B_I), (C, Y_I = 3 \rightarrow 8 \diamond C_I),$ $(D, Y_I = 4 \rightarrow 16 \diamond D_I)]_1$	`true`

Figure 7. A symbolic evaluation trace showing the stack top table and the current condition.

S contains one empty table.

When the first if-directive is met, the then-block is entered with the current condition $Y_I = 1$. Moreover, a new empty table is stacked onto S. At line 2, the variable A is bound to 2 in this new table.

At the exit of the if-directive, line 3, the top two tables are merged to form the new top table. In this example, the merging is simple since the old table is empty and the new table contains only one variable, namely A. From the old table, A has no value, that is its value is A_I, and in the top table it is 2. The if condition is $Y_I = 1$ so A has value 2 under that condition and the old value otherwise, symbolically $Y_I = 1 \rightarrow 2 \diamond A_I$.

When the second if-directive is met, a new empty table is pushed and the current condition becomes $Y_I = 2$. Then, the variable B is bound to 4 in the new top table. At the exit of the then-block, the top table is merged with the old table. This table contains the old binding for A and the c-value $Y_I = 2 \rightarrow 4 \diamond B_I$ for B.

Similar operations are done for C and D, resulting in a table of four bindings before considering the directive `#if defined(D)`. At this point, the c-value of D is $Y_I = 4 \rightarrow 16 \diamond D_I$. So, the condition is equivalent to $\texttt{def}?(Y_I = 4 \rightarrow 16 \diamond D_I)$ — which simplifies to $Y_I = 4 \vee \texttt{def}?(D_I)$. This is indeed the full condition for which the line `int x;` is compiled. That is, if the initial value of Y is 4 or the variable D is defined at the beginning of the preprocessing, then line 18 is compiled.

5.3. These examples show linear time complexity

As can be seen through these examples, our symbolic evaluation does not backtrack or consider several paths. It strictly proceeds forward by merging variable bindings using c-values. Although some if-directives are considered to evaluate variables A, B, and C, they do not substantially affect the efficiency to evaluate `#if defined(D)`.

Note also that even if there were a large number of if-directives, before the five directives considered, it would not exponentially increase the evaluation time or size of the c-values. The c-values length increase at most linearly with each new binding. It is also a direct consequence of the merging algorithm that if these directives do not create new bindings for Y or D, then the c-value of D is not affected in any manner. Therefore, these independent directives do not increase the evaluation time of the directive `#if defined(D)` — a necessity if a long series of if-directives is analyzed.

```
1. #if !defined(H)
2. #define H
3. ...
4. #include "g.c"
5. ...
6. #endif
```

```
1. #if !defined(G)
2. #define G
3. ...
4. #include "h.c"
5. ...
6. #endif
```

Figure 8. At left, file `h.c`, at right, file `g.c`; they depend on each other.

```
1. #if defined(F)
2. ...
3. #include "t.c"
4. ...
5. #endif
```

```
1. #if defined(T)
2. ...
3. #include "f.c"
4. ...
5. #endif
```

Figure 9. At left, file `f.c`, at right file `t.c`; there is no infinite loop if at most one is defined at preprocessing time.

6. Iterations

Although they are not explicitly provided by directives, iterations may occur in C/C++ preprocessing due to file inclusions. A common example is presented in Fig. 8: files `h.c` and `g.c` mutually includes each other. No infinite loop exists since guarded if-directives, based on variables G and H avoid multiple inclusions.

In general, any form of iteration is possible. Symbolic evaluation must properly handle all cases to avoid infinite loop or reporting an infinite loop when in fact none exist. As we will see, this is possible for preprocessing, even with free variables, since it comes down to determining if Boolean expressions are satisfiable or contradictory.

In the case of Fig. 8, it is easily handled since the symbolic evaluator evaluates the conditions as the preprocessor does: the variables G and H are defined in the top table and no multiple inclusion occurs. So, for example, when including file `h.c` for the first time, the conditional `#if !defined(H)` contains a free variable, but on the inclusion by `g.c` this condition becomes `false`.

On the other hand, there are other more complicated cases where conditions always contain free variables. These expressions can be satisfiable, yet when provided with specific values they become `false`. Therefore, we cannot report an infinite loop even though some lines of code (node in the CFG) are processed multiple times.

In C/C++ preprocessing, the number of nested file inclusion is limited. If the limit is k, this implies that no node of the CFG should be visited more than k times. This effectively gives symbolic evaluation an upper limit to stop all infinite iterations that may occur.

For example, consider Fig. 9 where an infinite iteration occurs when T and F are defined; but if only one is defined, there is no infinite loop, so symbolic evaluation cannot report one. Yet it must proceeds to properly infer the Boolean expression: To avoid the infinite loop, it limits to k the number of visit to any CFG node. These repeated visits may simply generate the same conditions. Once the disjunctive condition, formed from the list of these conditions, is simplified, the correct full condition is obtained.

7. Syntactical restrictions on if-directives

In general, an if-directive may have a non-expanded condition having no syntactical structure, but for which macro-expansion leads to a correct syntactical form. That is, prior to macro-expansion a condition is simply a list of tokens, but after macro-expansion it should have a syntactical structure of a Boolean condition. We restrict if-conditions to be such that macro-expansion does not depend on non evaluated conditions occurring in conditional values of substituted variables. These cases rarely occur; if they do, they point to obscure conditional compilations difficult to understand and maintain.

For example, Fig. 10 line 4 shows an if-directive that cannot be handled by our symbolic evaluation. To be handled by `cpp`, the variable `M` must be bound to tokens inducing a correct parsing of '`3 M 4`'. If it is undefined, the value of `M` is '`+ 1 ==`' and condition '`3 M 4`' becomes '`3 + 1 == 4`', which is `true`. But in general, symbolic evaluation does not know if `M` is defined and cannot parse the condition '`3 M 4`' properly, since some value of `M` may lead to a parsing error. On the other hand, if the symbolic evaluation can determine a base value for `M`, when reaching line 1, parsing of '`3 M 4`' becomes deterministic and a correct decision can be reached.

Cases involving macro with parameters can lead to complex conditional syntactical analysis. For example, Fig. 11 shows a case where the parsing of '`M ,50)`' not only depends on the value of `M`, but also on the macro called. The first `gcc` invocation, at line 9, makes the condition, on line 5, `true`. The second makes it `false` and involves a macro unknown in the source code. In general, it may be the case that `M` is initialized with a macro call completely unknown in the source code. In such cases, the conditions become not only arithmetical problems, but also syntactical combinatorial problems. Our symbolic evaluation cannot completely handled these cases although it detects them and pinpoints to bad preprocessing code that should be rewritten to become more understandable.

```
1. #ifndef M
2.   #define M   + 1 ==
3. #endif
4. #if 3 M 4
5.   int x = 10;
6. #endif
```

Figure 10. An example of an if-directive not handled by our symbolic evaluation.

```
1. #define X(x,y) x+y
2. #ifndef M
3.   #define M  X(20
4. #endif
5. #if M ,50) == 80
6.   int x = 10;
7. #endif
8.
9. gcc -D'M=X(30' f.c
10. gcc -D'M=Y(40' -D'Y(x,y)=x*y' f.c
```

Figure 11. Example using a macro with parameters and initializations on `gcc` invocations.

8. Experimental results

We have written an implementation of our algorithm in Scheme [5], a Lisp dialect, and applied it to some Unix C code. We have not made any effort to make it fast or space efficient. We have used a 930MHz Pentium III with 512MB of RAM computer, running Linux 2.2.12. All reported timing information comes from the Gambit [2] interpreter, version 3.0. That is, the symbolic evaluation implementation executes interpreted, not compiled.

8.1. A Java virtual machine implementation

Harissa [10] is an implementation of a Java Virtual Machine (JVM). Its main file `vm.c` includes four times the file `vm_exec.c`, each time by defining new preprocessing variables to tailor four different JVMs. It is an interesting case, as most nodes of the CFG are preprocessed four times with different variable values.

Parsing the 12420 lines (8227 are C lines) resulted in 3386 nodes in the CFG. It includes 72 files, 1291 definitions and 652 if-directives. Given this large number of conditional directives, any attempt to analyze all paths would take a huge amount of time.

Our prototype takes 6.4 seconds to symbolically evaluate this software, where most nodes are visited four times. There are 1012 preprocessing variables in the final table.

For example, line 1171 of `vm_exec.c`:

```
addCallProfile(m,pc-m->code-3,cm);
```

is reached under condition:

$$\texttt{def?}(\texttt{PROFILER}_I) \wedge \neg\texttt{def?}(\texttt{NO\_VM\_QUICK\_INSTRUCTIONS}_I) \wedge \neg\texttt{def?}(\texttt{VM\_NO\_QUICK}_I)$$

This condition is not apparent from the source code, since the variable `NO_VM_QUICK_INSTRUCTIONS` is used 1000 lines before this line to conditionally define `VM_NO_QUICK`; line 1171 is in turn conditionally compiled if this latter variable is not defined. This simple example shows that manually analyzing even simple conditional compilation may be very time consuming.

Since each CFG node contains the conditions of its traversal, it is possible not only to get the exact condition of reachability (by oring them), but also to analyze each individual case.

8.2. Linux header file `kernel.h`, version 2.2.12

The Linux kernel header file `kernel.h`, the old version 2.2.12, was symbolically evaluated. It directly includes files `linkage.h` and `stdarg.h`, which includes sixteen other files (e.g. `va-clipper.h`, `va-m88k.h`) defining various macros to handle variable number of arguments on different CPU architectures and software. The total number of lines is 2399 and the CFG has 828 nodes.

It takes 0.6 second for the symbolic evaluation of the CFG, which means that the exact Boolean conditions are derived for **all lines of all files**, not just one line, in less than one second. The final table contains exactly 100 preprocessing variable bindings.

In contrast, [4] finds one condition for one line at a time. We cannot directly compare our execution time with theirs, since they do not report any specific time for the full (exact) conditions found, although they report that:"The goal of finding the full condition to reach a tested code line seeks to cover all the combination of conditions, thus is of course time consuming." This is expected since their algorithm considers all paths. As a matter of fact, the file `stdarg.h` contains a nested series of more than twenty four if-directives which generates over four millions paths.

Here are three examples of line reachability conditions derived by our algorithm:

- Line 40 of file `kernel.h`

  ```
  #define FASTCALL(x) x
  ```

  is reached under condition

  $$\neg\texttt{def?}(\texttt{\_\_i386\_\_}_I) \wedge \texttt{def?}(\texttt{\_KERNEL\_\_}_I) \wedge \neg\texttt{def?}(\texttt{\_LINUX\_KERNEL\_H}_I)$$

  For the same line, [4] finds the full condition

  $$\neg\texttt{def?}(\texttt{\_\_i386\_\_}_I) \wedge \texttt{def?}(\texttt{\_KERNEL\_\_}_I)$$

since they assume the variable _LINUX_KERNEL_H as undefined before symbolic evaluation. In general, they consider all variables defined after being tested by #ifndef to be "safe guarders", which they assume undefined before symbolic evaluation. Our result is more complete, though, as it truly represent the exact condition to reach that line; in any case, if such results are expected, we could bind such variable to value $\perp$ before our symbolic evaluation and obtain the same result. Our approach is therefore more general.

- Lines 18 and 19 of file va-clipper.h:

```
#define va_list __gnuc_va_list
#define __va_list __gnuc_va_list
```

are reached under condition

$$(\neg\texttt{def?}(\texttt{\_\_need\_\_va\_list}_I) \vee \texttt{def?}(\texttt{\_VARARGS\_H}_I)) \wedge \texttt{def?}(\texttt{\_\_clipper\_\_}_I) \wedge \neg\texttt{def?}(\texttt{\_ANSI\_STDARG\_H\_}_I) \wedge \neg\texttt{def?}(\texttt{\_STDARG\_H}_I) \wedge \texttt{def?}(\texttt{\_\_KERNEL\_\_}_I) \wedge \neg\texttt{def?}(\texttt{\_LINUX\_KERNEL\_H}_I)$$

- Line 29 of file linkage.h:

```
#define __ALIGN .align 4
```

is reached under condition

$$\neg\texttt{def?}(\texttt{\_\_arm\_\_}_I) \wedge \neg\texttt{def?}(\texttt{\_LINUX\_LINKAGE\_H}_I) \wedge \texttt{def?}(\texttt{\_KERNEL\_\_}_I) \wedge \neg\texttt{def?}(\texttt{\_LINUX\_KERNEL\_H}_I)$$

9. Related works

Livadas and Small [9] have built a GUI tool to slice parts of C source code, taking into account preprocessing variables and directives, but it does not infer the Boolean expressions for line reachability.

Krone and Snelting [7] based their graph display of configuration structures on C preprocessor directives. They mainly focus on the overall software structure. They do not take into account variables set inside #ifdef; therefore they do not infer the complete Boolean expressions for each line of code.

Harsu [3] presents a technique to translate source code containing preprocessor directives. The author pinpoints the difficulty of processing source files with numerous free preprocessing variables; even when automatic translation is involved.

```
1.  #if !defined(W) && defined(Z)
2.   z = z + 1;
3.  #endif
4.
5.  #if defined(W) && !defined(Z)
6.   w = w + 1;
7.  #endif
8.
9.  #if defined(W) && defined(Z)
10.  w = w + z;
11. #endif
```

Figure 12. A series of three if-directives generate eight paths.

Kullbach and Riediger [8] propose a folding technique to hide some parts of the source code based on the preprocessing directives. This technique addresses the problem of maintenance of code with preprocessing directives, although it is mainly a visual approach.

The work of Hu *et al.* [4] addresses very similar problems of this paper. They offer two techniques, one to find the "simplest sufficient condition to reach/compile" a line and another for the "full condition to reach/compile" it. Their approach uses the common technique of symbolic evaluation using paths, pioneered by [6].

Their first technique uses the shortest path. We believe this has a major flaw, since the path found can contain a series of contradictory constraints. It is a well known fact that symbolic evaluation using paths can go through a series of constraints that are contradictory and it is quite common to go through such paths[1]. These are infeasible path and symbolic evaluation by itself does not identify them. Giving the shortest path, and not verifying that its constraints are not contradictory, can provide an incoherent condition to reach that line. Such a condition can hardly be useful since another path can be non-contradictory.

It is also very debatable if a shortest path condition is useful, since there are a large number of paths having the same length. Consider Fig. 12, a series of three conditional. There are eight shortest paths, from line 1 to 10, of length three: Seven are contradictory and only one is not, namely the one that assumes the first two conditions are `false` and the third is `true`. So, a path simply uncovers a series of constraints, possibly contradictory, under which the line is reachable. Given the numerous shortest paths to reach a line, it is very difficult for the programmer to give any precise meaning to one of them. It is actually necessary to have a theorem prover to derive non-contradictory paths. But that problem is not addressed in their paper since simplifications of conditions is done in a last phase by an external simplifier detached from the symbolic evaluator.

The second technique uncovers all paths, in the CFG, to one line under analysis, to derive the full (exact) condition

of reachability. (This is one of the problems adressed in this paper.) As discussed in sections 1 and 8, this technique is very inefficient, since there are 2^n paths for a series of n (not necessarily nested) if-directives. A series of more than thirty if-directives is common, resulting in billions of paths. Indeed, the authors do not report any evaluation time for the full conditions found.

10. Conclusion and future work

It has been shown that symbolic evaluation of C/C++ preprocessing can be done efficiently by using conditional values (c-values). The conventional symbolic approach of analyzing execution paths has been shown to be of exponential time complexity in the best case; whereas using c-values, it has a linear time complexity for that case.

This evaluation gives, for each line of code, a symbolic conditional expression determining its reachability and the conditional value of each preprocessing variable. This fundamental information can be used to do further code analysis.

C/C++ preprocessing has an important constraint, namely the limit of nested file inclusions, that makes it possible to avoid infinite iteration. Without this constraint, iteration analysis would become undecidable.

Experiments on some Unix software have shown the practicality and efficiency of our symbolic algorithm. It was specifically shown to be much faster than traditional symbolic evaluation by avoiding exponentional time complexity.

Simplification of conditional values is an important aspect of our approach; without them, determining conditional expression status (e.g. satisfiable, tautology, contradictory) could be time consuming.

Our set of simplification rules is based on our common sense and experiments — not all reported in this paper. This set could be incomplete to efficiently tackle some cases. In this regard, future work may provide an expanded set of useful simplification rules.

References

[1] P. D. Coward. Symbolic execution systems – a review. *Software Engineering Journal*, pages 229–239, November 1988.

[2] M. Feeley. Gambit Web Page. URL: www.iro.umontreal.ca/~gambit/.

[3] M. Harsu. Translation of conditional compilation. *Nordic Journal of Computing*, 6(1), Spring 1999.

[4] Y. Hu, E. Merlo, M. Desmarais, and B. Lagüe. C/C++ conditional compilation analysis using symbolic execution. In *Proceedings of the international Conference on Software Maintenance (ICSM'00)*, 2000.

[5] R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[6] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[7] M. Krone and G. Snelting. On the inference of configuration structures from source code. Technical Report 93-06, Gausstrasse 17, D-38092 Braunschweig, Deutschland, 1994.

[8] B. Kullbach and V. Riediger. Folding: An Approach to Enable Program Understanding of Preprocessed Languages. Fachberichte Informatik 7–2001, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.

[9] P. Livadas and D. Small. Understanding code containing preprocessor constructs. In IEEE Third Workshop on Program Comprehension, pages 89–97, Washington, DC, USA, November 14–15, 1994.

[10] G. Muller and U. P. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, 16(2):44–51, 1999.

[11] H. Spencer and G. Collyer. ifdef considered harmful, or portability experience with c news. In *Summer '92 USENIX*, pages 185–198, June 1992.