

# Conditioned Slicing

David Jong-hoon An

May 23, 2007

## Abstract

Program debugging is often a tedious and difficult process that requires programmers to inspect complicated code to understand and analyze the program for correctness. In general, programmers spend much time determining which lines of code may directly or indirectly cause a program crash or a wrong output. Program slicing is a technique that automates this task by projecting only the subset of the program that may affect the point of interest which could be a crash point or a point where an incorrect value is printed out. Because the irrelevant code is sliced out, it reduces the amount of time for debugging or other analysis.

One potential problem with program slicing, however, is that the slices can be too large to be of much use. In this thesis, we discuss a technique called conditioned slicing which takes into account additional constraints to produce smaller slices. We improve this technique by considering data dependence edges in addition to program components when removing elements from the program dependency graph. In this thesis, we also present our implementation of a conditioned slicing tool for the C language and additional improvements that can be done in the future.

## 1 Introduction

Program debugging is often a tedious and difficult process that requires programmers to inspect complicated code to understand and analyze the program for correctness. In general, programmers spend much time determining which lines of code may directly or indirectly cause a program crash or a wrong output. Program slicing is a technique that automates this task by projecting only the subset of the program that may affect the point of interest (POI) which could be a crash point or a point where an incorrect value is printed out. Because the irrelevant code is sliced out, it reduces the amount of time for debugging or other analysis.

Consider the example in Figure 1a which calculates Federal tax based on user inputs. Let us assume that `printf("%d", deducts)` at line 14 is our POI, presumably because the value of `deducts` is incorrect at that point. Figure 1b shows

Figure 1: Federal tax program and its slices with respect to two different POIs

Original Program	Slice with Respect to Line 14	Slice with Respect to Line 15
<pre> 1: scanf("%d", &amp;charitableGifts); 2: scanf("%d", &amp;stateTaxPaid); 3: scanf("%d", &amp;wages); 4: scanf("%d", &amp;otherInc); 5: deducts = charitableGifts 6:     + stateTaxPaid; 7: income = wages + otherInc 8:     - deducts; 9: if (income &lt;= MAX) 10:     taxes = Taxtable(income); 11: else 12:     taxes = AltTax(income); 13: printf("%d", income); 14: printf("%d", deducts); 15: printf("%d", taxes); </pre>	<pre> 1: scanf("%d", &amp;charitableGifts); 2: scanf("%d", &amp;stateTaxPaid); 5: deducts = charitableGifts 6:     + stateTaxPaid; 14: printf("%d", deducts); </pre>	<pre> 1: scanf("%d", &amp;charitableGifts); 2: scanf("%d", &amp;stateTaxPaid); 3: scanf("%d", &amp;wages); 4: scanf("%d", &amp;otherInc); 5: deducts = charitableGifts 6:     + stateTaxPaid; 7: income = wages + otherInc 8:     - deducts; 9: if (income &lt;= MAX) 10:     taxes = Taxtable(income); 11: else 12:     taxes = AltTax(income); 15: printf("%d", taxes); </pre>
(a)	(b)	(c)

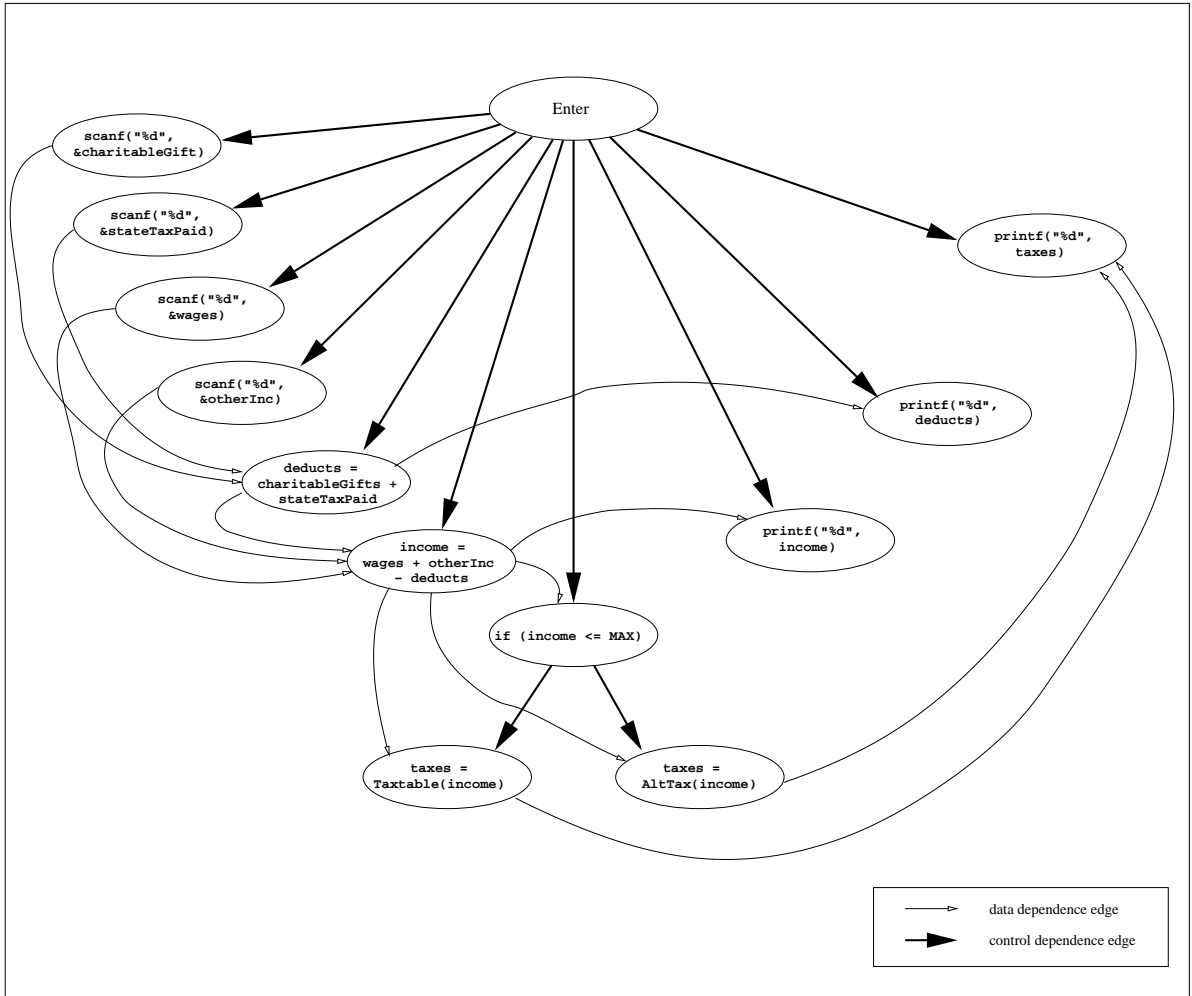
that the slice of the example with respect to the POI is only a small part of the program which helps programmers to focus only on the code that influences the incorrect output rather than the entire program.

The initial slicing technique done by Mark Weiser [2] constructs slices with respect to a slicing criterion,  $\langle p, V \rangle$ , where  $p$  is a program point and  $V$  is a set of variables of interest. Any program components that do not transitively influence  $V$  at  $p$  are excluded from the slice. In other words, the slice includes only the subset of a program that satisfies the following: It affects (1) the value of variables used at the POI or (2) the number of times that the POI or the statements influencing the POI execute when the program is run.

Slicing can be done efficiently by representing the program as a program dependency graph [1] in which the vertices represent program components and the edges represent control and data dependences. Using the graph, the slicing problem now becomes a graph-reachability problem. There exists a control dependence from  $n$  to  $m$  if the execution of  $m$  is controlled by  $n$  such as in an `if` statement. There is a data dependence between  $n$  and  $m$  if  $n$  writes a value that may be read by  $m$ . Figure 2 shows the graph representation for the example program of Figure 1. Note that the slice of the program with respect to `printf("%d", deducts)` can be obtained by solving a backward graph-reachability problem for the corresponding node.

One potential problem with slicing, however, is that slices can sometimes be too large to be of much use. For example, let us assume that our POI is `printf("%d", taxes)` at line 15 of the program in Figure 1a (presumably because it had a wrong value of `taxes` printed out). Figure 1c shows the slice obtained with respect to the new POI, which includes the whole program except two `print` statements. This is because all the statements in the slice may

Figure 2: Program dependency graph of the code in Figure 1



affect the value of taxes at line 15. Therefore, in this case, slicing does not provide any help for debugging the program.

At the point of failure, however, there might be useful information, such as the values of some variables at that point, which may be exploited to reduce the size of the slice. A technique called *conditioned slicing* [3] takes into account these conditions as constraints on the slicing criterion. For example, let us assume that for our example program we know that when `printf("%d", taxes)` is executed, `income < MAX`. This information can be used to determine that the *else* branch of the `if` statement at line 9 has never been executed—i.e., the statement at line 12 is only on infeasible paths. Therefore, the statement at line 12 can be excluded from the slice.

We can improve the conditioned slicing technique defined in [3] by taking into account data dependences that are only induced by infeasible paths. For instance, Figure 3a shows a slightly different version of the example program shown earlier. This program always executes the statement `taxes = AltTax(income)` at line 9; thus, the statement is on all feasible paths to the statement `printf("%d", taxes)` at line 14. However, if condition `income < MAX` holds at line 14, then the statement, `taxes = AltTax(income)` would not affect the printed value because `taxes` is guaranteed to be overwritten by the statement `taxes = Taxtable(income)` at line 11. Therefore, the statement at line 9 can be excluded from the slice although it is always executed.

This improvement can be accomplished by removing the data dependences that are induced only by infeasible paths in the context of the given condition. For example, the data dependence from `taxes = AltTax(income)` to `printf("%d", taxes)` is induced only by the *else* branch of the `if` statement. This branch, however, is only on an infeasible path because the given condition, `income < MAX`, is consistent with the condition of the `if` statement, `income <= MAX`. By removing the data dependence, the slice without the statement `taxes = AltTax(income)` will be computed.

The remainder of the thesis is organized as follows: Section 2 gives an overview of our improved conditioned slicing technique along with some examples. Section 3 describes the implementation details of our conditioned slicing tool for the C language. Section 4 provides possible improvements that can be done as future work. Finally, Section 5 concludes the thesis.

## 2 Overview

Conditioned slicing is done in five phases: partitioning, writing symbolic formulas, theorem proving, pruning, and slicing. Initially, a program is partitioned into a set of paths so that we can generate symbolic formulas to represent states along each path. Each path is represented with a sequence of CFG nodes. Since loops and recursion cause an infinite number of paths, we must approximate cyclic paths by “collapsing” the CFG nodes into a single node. For

Figure 3: Variation of federal tax program and its slices with respect to two different POIs

Original Program	Previous Conditioned Slicing	Our Improved Conditioned Slicing
1: scanf("%d", &charitableGifts);	1: scanf("%d", &charitableGifts);	1: scanf("%d", &charitableGifts);
2: scanf("%d", &stateTaxPaid);	2: scanf("%d", &stateTaxPaid);	2: scanf("%d", &stateTaxPaid);
3: scanf("%d", &wages);	3: scanf("%d", &wages);	3: scanf("%d", &wages);
4: scanf("%d", &otherInc);	4: scanf("%d", &otherInc);	4: scanf("%d", &otherInc);
5: deducts = charitableGifts	5: deducts = charitableGifts	5: deducts = charitableGifts
6:       + stateTaxPaid;	6:       + stateTaxPaid;	6:       + stateTaxPaid;
7: income = wages + otherInc	7: income = wages + otherInc	7: income = wages + otherInc
8:       - deducts;	8:       - deducts;	8:       - deducts;
9: taxes = AltTax(income);	9: taxes = AltTax(income);	
10: if (income <= MAX)	10: if (income <= MAX)	10: if (income <= MAX)
11:     taxes = Taxtable(income);	11:     taxes = Taxtable(income);	11:     taxes = Taxtable(income);
12: printf("%d", income);		
13: printf("%d", deducts);		
14: printf("%d", taxes);	14: printf("%d", taxes);	14: printf("%d", taxes);
(a)	(b)	(c)

each path obtained, a set of symbolic formulas is written out to represent the values of the variables at each point in the path. In actual executions of the program, variables are always assigned actual values, whereas in symbolic formulas, the variables are assigned constant values, symbolic variables, or a function of these elements.

In addition to the assignments, each set of formulas has assertions that say certain conditions are true (or false) at control-points of the program. The symbolic formulas do not provide concrete information about the variables at the final state (the POI), but we can use a theorem prover to compute the satisfiability of each set of formulas and determine the feasibility of the corresponding path with respect to the assertions and user conditions (an input to the slicing tool). Based on the results of the theorem prover, we prune appropriate nodes and edges from the PDG. First, we remove all program components that occur only in infeasible paths from the PDG. This process ensures that these components are not included in the slice. Second, data dependence edges that are induced only by infeasible paths are excluded from the PDG to refine the slicing at the final phase.

Consider the simple example in Figure 4a, which is a subset of the example in Figure 3a with a slight variation. The call to `scanf()` at line 1 is treated as a special case that assigns an unknown value to its argument. The other function calls are regarded as typical expressions involving the associated arguments (basically, ignoring interprocedural flow edges). There are two *possible* paths in this program—one taking the *then* branch and another one taking the *else* branch of the `if` statement at line 3. In the partitioning phase, the program is divided into these paths which would be represented as  $(n_1 n_2 n_3 n_4 n_5)$  and  $(n_1 n_2 n_3 n_5)$  where  $n_i$  represents a CFG node corresponding to line  $i$ . For each of these paths, a set of symbolic formulas is written out as shown in the example.

There are two kinds of statements in symbolic formulas: assignments and assertions. An assignment assigns a

Figure 4: Simple version of Federal tax program

Original Program	Symbolic Formulas for Path 1	Symbolic Formulas for Path 2
1: scanf("%d", &income);	$income_1 = userInput_1$	$income_1 = userInput_1$
	$taxes_1 = taxes_0$	$taxes_1 = taxes_0$
	$return_1 = return_0$	$return_1 = return_0$
2: taxes = AltTax(income);	$income_2 = income_1$	$income_2 = income_1$
	$taxes_2 = AltTax(income_1)$	$taxes_2 = AltTax(income_1)$
	$return_2 = return_1$	$return_2 = return_1$
3: if (income <= MAX)	$assert(income_2 \leq MAX)$	$assert(\neg(income_2 \leq MAX))$
4:     taxes = Taxtable(income);	$income_3 = income_2$	
	$taxes_3 = Taxtable(income_2)$	
	$return_3 = return_2$	
5: return taxes;	$income_4 = income_3$	$income_3 = income_2$
	$taxes_4 = income_3$	$taxes_3 = taxes_2$
	$return_4 = taxes_3$	$return_3 = taxes_2$

(a)
(b)
(c)

function of constants and previously defined symbolic variables to a newly declared symbolic variable, which changes the current state of the program execution. Note that all used variables in the program have corresponding symbolic variables that are newly defined at each assignment, as illustrated in Figure 4b and Figure 4c. This is to maintain all symbolic variables at each assignment of the program execution.

Unlike the assignments, assertions do not change the state of the program execution, but rather, they provide useful information about conditions satisfied by the paths, starting from the entry point to the POI. For example, Path 1 has taken the *then* branch and Path 2 has taken the *else* branch of the `if` statement, and each of them has its condition (either the conditional expression or its negation) upheld in the corresponding `assert` statement. The theorem prover, then, computes the satisfiability of the set of formulas based on the consistency of these assertions and a given condition. If `income` at line 1 is greater than `MAX`, the set of formulas for Path 1 would be inconsistent, which is equivalent to saying Path 1 is *infeasible*.

Next, in the pruning phase, we exclude from the PDG program components and data dependence edges based on the information about infeasible paths. To do this, we first determine which program components occur only in infeasible paths. For example, let us assume that Path 1 is *infeasible* and Path 2 is *feasible*. We can determine that `taxes = Taxtable(income)` at line 4 occurs only in Path 1 which is infeasible. Therefore, it can be removed from the PDG. Figure 5 shows the PDG with the node corresponding to line 4, denoted by (1). Removing (1) ensures that it is not included in the slice when we do a slicing on the remaining graph in the final phase.

As mentioned earlier, we have improved conditioned slicing by eliminating data dependence edges that are induced only by infeasible paths. For example, let us now assume that Path 1 is *feasible* and Path 2 is *infeasible*. Note that the data dependence edge between `taxes = AltTax(income)` and `return taxes` which is denoted by (a) is induced

Figure 5: Program dependency graph of the example in Figure 4a

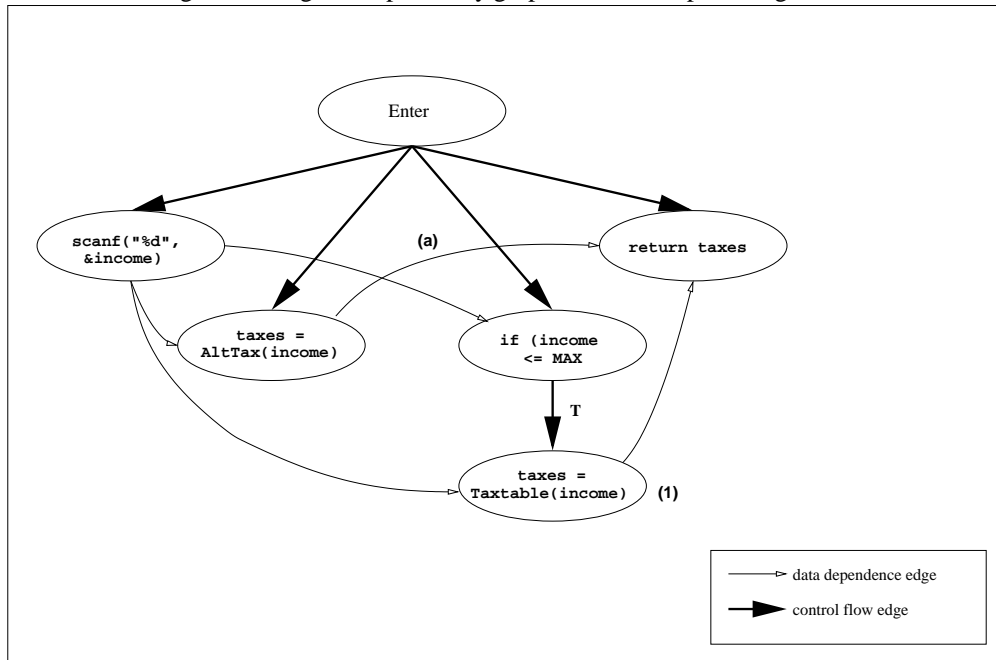


Figure 6: Program dependency graph after removing (a).

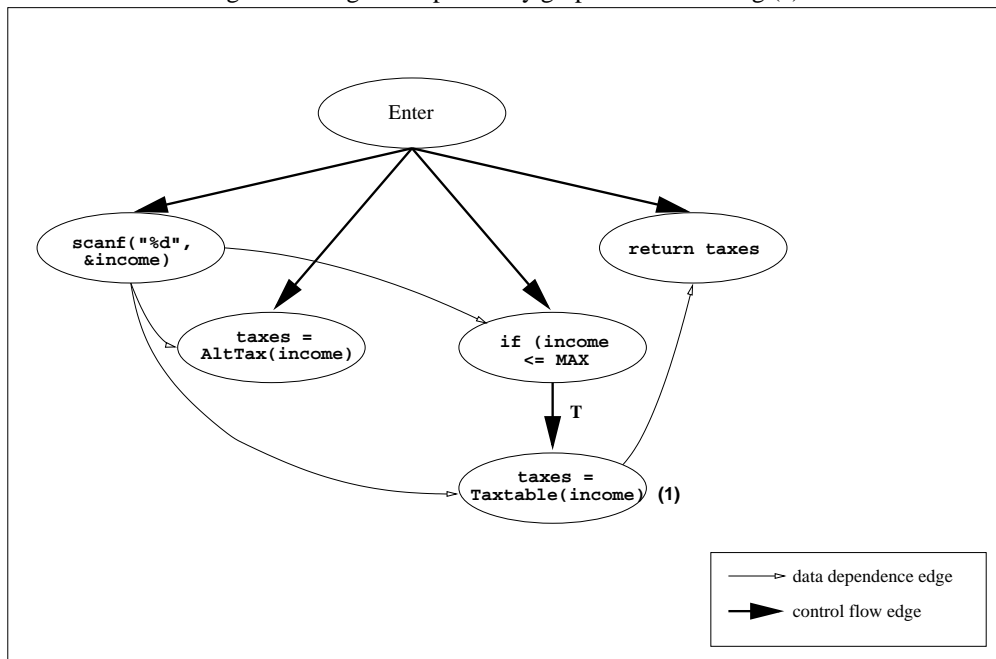
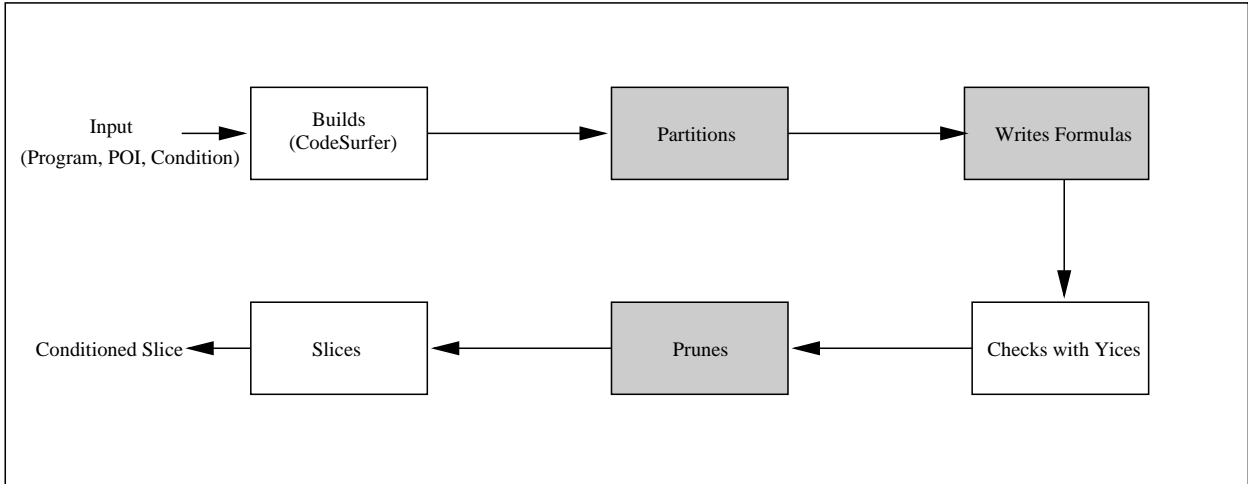


Figure 7: Structure of our conditioned slicing tool



only by Path 2, the infeasible path. Figure 6 shows the PDG after removing the edge (a). When a slicing is performed on the remaining graph with respect to line 5, the statement `taxes = AltTax(income)` will be excluded from the slice since its corresponding node is not reachable from the node corresponding to the POI.

Note that a slice alone would include everything because all nodes in the PDG are reachable going backward from the node corresponding to `return taxes`. Conditioned slicing does a better job because it takes into account additional constraints given by the user, such as `income <= MAX` on the slicing criterion to produce a smaller slice. The next section discusses more details about how we implemented the conditioned slicing tool for the C language and illustrates the tool with several examples.

### 3 Implementation

This section describes the implementation details as well as some difficulties and challenges during the implementation. Section 3.1 gives an overview of our conditioned slicing tool, and Section 3.2 defines the input to the tool. Section 3.3 explains the partitioning process which divides a program into a set of acyclic paths, and Section 3.4 discusses how a set of symbolic formulas is written out for each path and explains in more detail about the formulas.

#### 3.1 General Structure

The structure of our conditioned slicing tool is illustrated in Figure 7, which shows the components that we implemented in shaded boxes. CodeSurfer [6] builds a project out of a source program and constructs corresponding CFGs and PDGs. CodeSurfer is a software analysis tool developed by GrammaTech that does static program slicing on C

programs. In addition, it provides useful application programming interfaces (APIs) to access information about the program such as the CFG and the PDGs. Our conditioned slicing tool is mostly implemented in Scheme using the CodeSurfer API for the Scheme language.

Once the CodeSurfer project is built, a program is partitioned into a set of acyclic paths using the CFG constructed by CodeSurfer. For each of these paths, a set of formulas is written so that they can be verified by Yices, the theorem prover used in our implementation [7]. The Yices theorem prover solves each set of formulas for its satisfiability which tells us the feasibility of the corresponding path. Based on this information, the appropriate nodes and edges are removed (pruned) from the PDGs. Finally, we perform slicing on the remaining PDGs to obtain a conditioned slice.

### 3.2 Input

The input to the conditioned slicing tool consists of a source program, the point of interest (POI) in that program, and user-supplied conditions that constrain the slicing criterion. Our tool accepts any C program that is written in ANSI C Standard. The POI may be a crash point or where an incorrect output is printed, which may be determined by a user or a debugger. Its format should be a source file name and a line number in that file whose corresponding CFG node(s) has a *single* predecessor.

The user-supplied conditions are used as additional assertions in each set of Yices formulas generated later. Each condition consists of a source file name, a line number in that file, and a conditional expression. For each variable used in the condition, either it must occur on the given line, or there can only be one declaration of that variable name in the whole program. This is to avoid the ambiguity of identical variable names declared in more than one scope. Furthermore, the CFG node(s) that correspond to the given line number must occur at most once on each path since it is not clear how to handle an assertion in a loop or recursion.

### 3.3 Partitioning

In the partitioning phase of conditioned slicing, a program is divided into a set of acyclic paths where each path is represented by a sequence of CFG nodes. The purpose of obtaining acyclic paths is to write a set of symbolic formulas that conservatively approximates the executions of all possible paths of the program including loops and recursions which cause cycles in the CFG. Therefore, prior to actually generating the paths, these cycles must be identified and “collapsed” to ensure the paths are acyclic. Once these components are all identified, we generate the acyclic paths of the program using depth-first search (DFS) on the CFG.

Figure 8: Example code that uses for loops

```
1: int f() {
2:   int sum = 0;
3:   for (int i = 0; i < MAX; i++) {
4:     for (int j = 0; j < MAX; j++) {
5:       sum += i + j;
6:     }
7:   }
8:   return sum;
9: }
```

### 3.3.1 Loops and Recursion

The CFG nodes of a loop are *strongly-connected* to each other because for every pair of nodes  $i, j$  in the loop there is a path from node  $i$  to node  $j$  and another path from node  $j$  to node  $i$ . This is illustrated in Figure 9 which shows the CFG of the example code in Figure 8. Note that nodes  $d, e, f, g, h,$  and  $i$  are strongly connected to each other: there is a path from any node to all other nodes among the strongly connected nodes. The maximal sub-graph that contains all nodes that are strongly connected to each other is called a *strongly-connected component* (SCC). For instance, nodes  $d, e, f, g, h,$  and  $i$  in the example form a single SCC that contains the bodies of both the inner and outer `for` loops. The SCCs are easy to compute and sufficient to fulfill the purpose of approximating all cyclic paths associated with loops.

The following algorithm [5] finds the set of SCCs in an intraprocedural CFG called  $G$ :

1. Let  $S$  be an empty set, and let  $L$  be a list that contains all nodes in  $G$ .
2. Let  $v$  be the exit node of the graph  $G^T$ , a transposed graph of  $G$ .
3. Do a depth-first search on  $v$ , labeling each node with its post-order number.
4. Let  $w$  be the node in  $L$  with the highest post-order number.
5. Let  $R$  be the set of nodes in  $G$  that are reachable from  $w$ .
6. If  $R$  has more than 1 node, add  $\{R\}$  to  $S$ .
7. Remove all nodes in  $R$  from  $L$ .
8. If  $L$  is empty, return  $S$ . Otherwise go to Step 4.

Note that this algorithm is only appropriate to *intraprocedural* CFGs of the program. Applying the algorithm to the *interprocedural* CFGs may calculate a wrong set of SCCs. The example C code in Figure 10 is used to illustrate

Figure 9: Example of a CFG with for loops

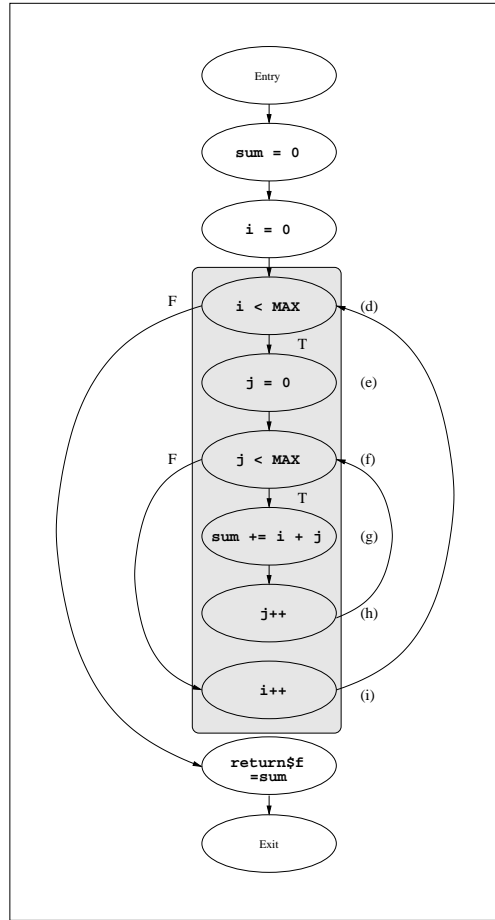
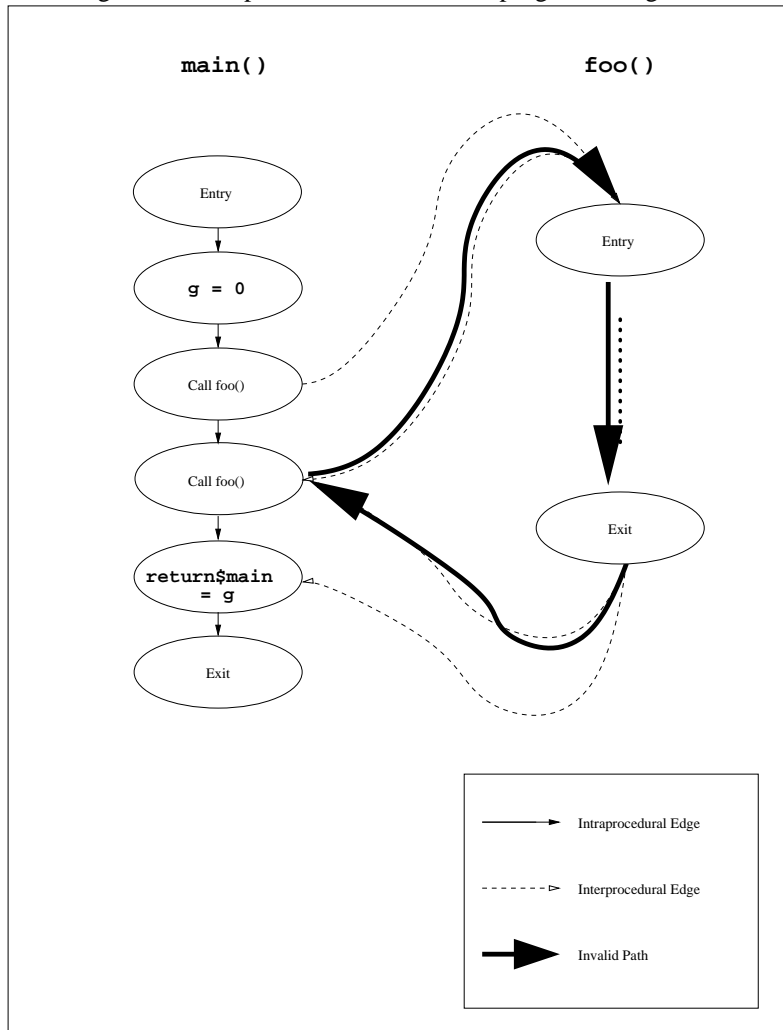


Figure 10: Example code to illustrate why SCCs must be computed using intraprocedural, not interprocedural CFGs

```
1: int g;  
2: void foo() {  
3:     // modifies g  
4: }  
5: int main() {  
6:     g = 0;  
7:     foo();  
8:     foo();  
9:     return g;  
10: }
```

Figure 11: Interprocedural CFG for the program in Figure 10



this problem. There are two functions `main` and `foo` and two calls to the function `foo` in the function `main`. We can clearly see that there are no cycles in this program. However, its interprocedural CFG does have a cycle, as shown in Figure 11, which is indicated by the bold directed lines. Note that this is an invalid path in the program; yet, the algorithm that finds SCCs cannot determine the validity of the path. To solve this problem, we simply prohibit visiting nodes through interprocedural edges and calculate SCCs for each intraprocedural CFGs instead. This solution, however, does not solve the problem of finding cycles due to recursion.

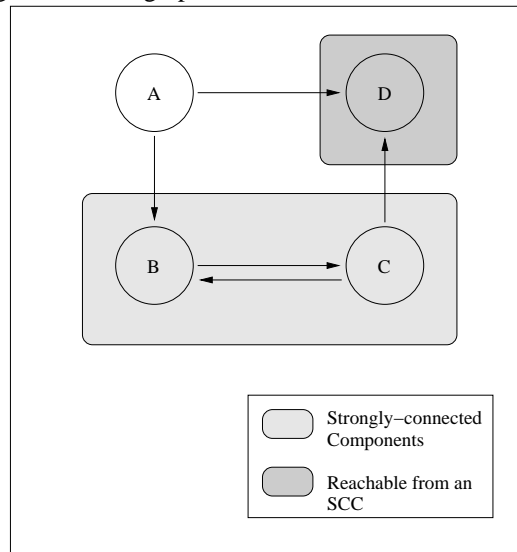
Recursion is a programming technique in which a function calls itself (or another function that calls it) to perform an operation in terms of itself. Since recursion occurs across function calls, cycles due to recursion can be computed using a call-graph that consists of nodes that represent functions, and edges that represent function calls in a program. Once the call-graph is constructed, this problem becomes another SCC finding problem. CodeSurfer actually constructs a call-graph for the program and also computes SCCs within the graph during a CodeSurfer project build. For some reason, CodeSurfer does not distinguish between a non-SCC and an SCC with one node in a call-graph. In other words, call-graphs created by CodeSurfer do not have any edge that leads out of and into the same node (for recursion that occurs within a single function). Therefore, we must check whether a function that CodeSurfer identifies as a single-node SCC actually is an SCC that represents a single-function recursion.

Recall that the purpose of collapsing cyclic paths is to conservatively approximate all possible paths in a program. An SCC does not include all the CFG nodes of the associated path because it may have calls to a function that is not a part of that particular SCC. Therefore, we must keep this extra information along with each SCC so that we can correctly represent all paths associated with that SCC. We represent this part of an SCC as a list of CFG nodes with no particular order.

Consider the example call-graph in Figure 12 where there is recursion between functions B and C, forming an SCC. Note that the function D is called from the function C but is not a part of the SCC because it does not have any edge leading into the SCC. However, all the CFG nodes of function D may be on a path that includes a node in the SCC because D may be called from that SCC. We cannot simply include the CFG nodes of the function D in the SCC because it is not part of the SCC (e.g., a path from function A to function D should not include the SCC). Thus, associated with the SCC, we keep a list of “SCC-reachable nodes”: the CFG nodes in the functions that are reachable from the SCC. For the call-graph in Figure 12, the SCC and “SCC-reachable nodes” would be  $((b_1 \dots b_i c_1 \dots c_j) (d_1 \dots d_k))$ , where  $b_1 \dots b_i$ ,  $c_1 \dots c_j$ , and  $d_1 \dots d_k$  refers to all the CFG nodes in B, C, and D, respectively.

The CFGs shown in Figure 13 illustrate a program that has a function call from an SCC due to a loop. The SCC and the “SCC-reachable nodes” obtained for this example would be  $((cde)(hij))$ .

Figure 12: Call-graph that has a function call in an SCC



### 3.3.2 Path Generation

Once loops and recursion are all identified as SCCs as well as the CFG nodes in the functions that may be called from nodes in the SCCs (but are not part of the SCCs), a program can be partitioned into a set of acyclic paths from the entry node of the function `main` to the POI. Another depth-first search algorithm is used, but going backwards, starting from the POI rather than the entry node because starting from the entry node may result in too many paths that never reach the POI. Consider the CFG shown in Figure 14 which has three possible paths from the entry node to the exit node. However, it has only one path from the entry node to the CFG node corresponding to the POI. Applying the algorithm backwards, starting from the POI, prevents visiting unnecessary CFG nodes and generating useless paths.

Recall that SCCs and the CFG nodes reachable from the SCCs must be treated conservatively to approximate all possible paths. Therefore, we include in each path the nodes associated with the SCCs that are part of the path so that we can access information about all of the CFG nodes on each path. For example, the SCC information obtained from Figure 13 is  $((cde)(hij))$ . When any of the nodes  $c, d$ , or  $e$  is encountered during the path generation, the list of all CFG nodes associated with the SCC is written out, which is  $(cdehij)$  in this example.

At the end of the algorithm, we end up with a list of path lists where each path list consists of CFG nodes or lists of CFG nodes representing SCCs; for example, for the CFG in Figure 13, we would have one path, represented as  $((ab(cdehij)fg))$ .

Figure 13: CFG that has a function call in an SCC

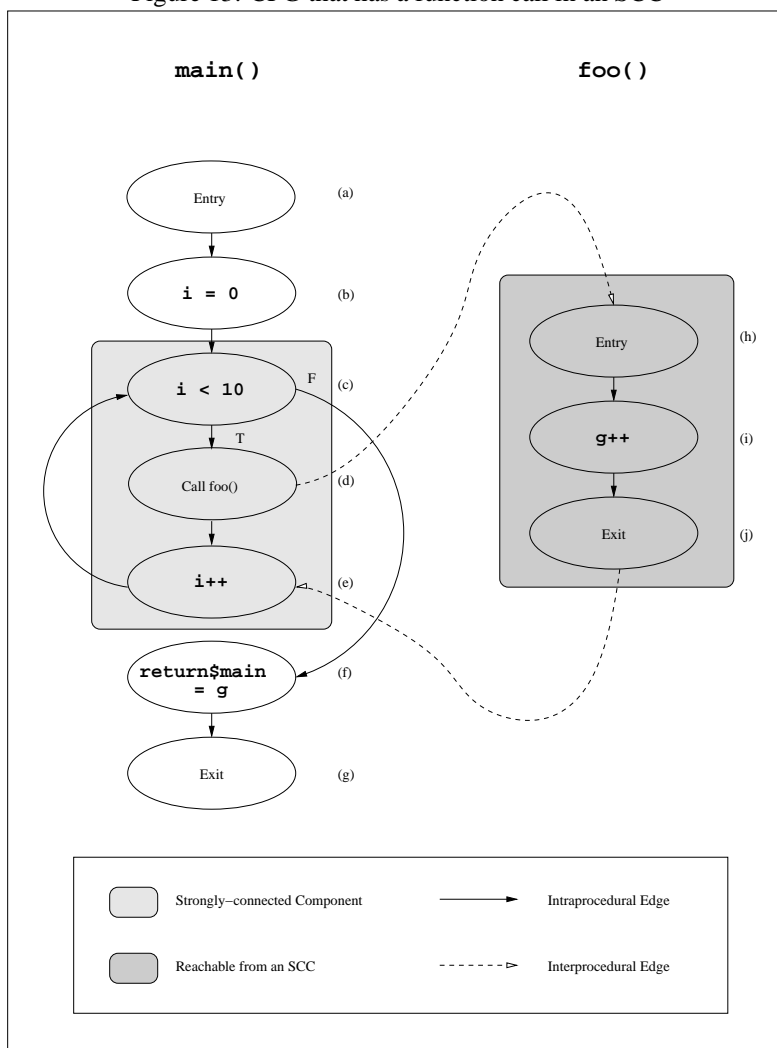
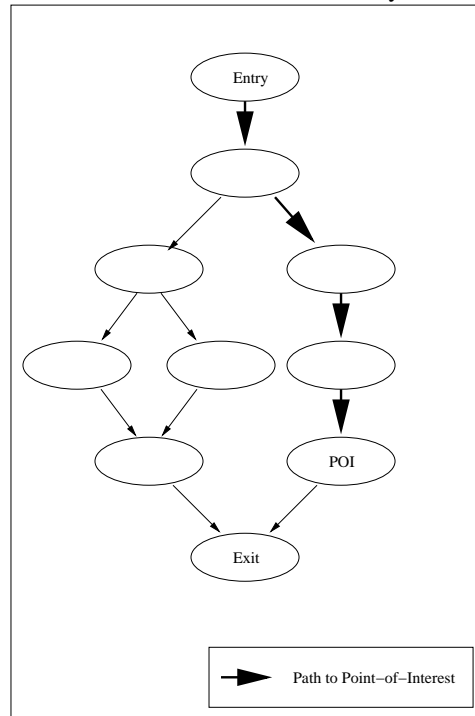


Figure 14: CFG with the POI that is only on one path



### 3.4 Yices Formulas

Yices formulas are written out for each path identified by the path-generation algorithm so that the Yices theorem prover can solve their satisfiability which also determines the feasibility of the path. There are two kinds of formulas in Yices: definitions and assertions. A definition defines a Yices variable that represents a C variable, and an assertion says that a certain condition is true. User-supplied conditions, one of the inputs to the tool, are added to the formulas as additional assertions at appropriate places. The Yices theorem prover solves each set of formulas for its consistency with respect to the assertions. If a set of formulas is consistent, the corresponding path is feasible. Otherwise, the corresponding path is infeasible.

Yices formulas use prefix notation, and Yices variables are defined as constants, meaning that they are assigned a value only once. Furthermore, each Yices variable must have a primitive type such as `int` or `real` which correspond to integer types and floating-point types, respectively, or a composed type such as `record` for structures and unions. More details on handling these types are discussed later in this section.

Our tool defines a set of Yices variables at declarations and every assignment statement in the program for every variable used in the program. The first set of Yices variables, which correspond to the variable declarations, is defined yet uninitialized, meaning that they are declared but not assigned any values (therefore, hold unknown val-

Figure 15: C code to illustrate Yices formulas

```
1: int radius, compute;
2: float result;
3: radius = 2;
4: compute = AREA; // defined 1
5: if (compute == AREA)
6:     result = 3.14 * radius * radius;
7: else // compute == CIRCUMFERENCE
8:     result = 2 * 3.14 * radius;
```

ues). This is done by using the Yices *define* operation with no right-hand side expression—e.g., the Yices statement `(define x_1::int)` declares `x_1` and assigns an unknown value to `x_1`. Note that the initialization of a variable at the declaration is normalized by CodeSurfer into a declaration and a separate assignment to the declared variable.

For each assignment after declarations, a set of Yices variables are defined with right-hand side expressions, meaning that they are declared and assigned some value. For example, the Yices statement `(define x_2::int y_1)` declares `x_2` and assigns `y_1` to `x_2`. The right-hand side expressions can be a previously defined variable, a constant, or a function of these elements. (If the right-hand side expression is not given, the define operator assigns an unknown value to the target variable.) We use a subscript to distinguish the Yices variables at the different assignments. Once a Yices variable is defined, it cannot be defined again but can be read anytime.

Consider the example code in Figure 15 which calculates the area or the circumference of a circle. (It actually computes the area of the circle because the flag `compute` is assigned `AREA` explicitly at line 4.) There are two sets of Yices formulas since there are two possible paths in the program. Each set of formulas has an assertion that corresponds to the *then* or the *else* branch of the `if` statement at line 5, as shown in Figure 16. For the variable declarations, a set of *define* operations is written to declare the corresponding Yices variables (i.e., to define them as having unknown values). The Yices formulas for the assignment to `radius` at line 3 consist of *define* operations for a new set of Yices variables that correspond to all the variables in the program. This is illustrated in the Yices formulas shown in Figure 16. Note that the Yices variable `radius_2` is assigned a constant 2 because it corresponds to the C variable `radius` at line 3. Other Yices variables, `compute_2` and `result_2` are defined as previously defined variables since no changes are done to the corresponding C variables here.

There are several kinds of variables in the C language to consider because they are all represented using different Yices *define* operations. The Boolean type is not supported by the C language. Instead, integer values are used to represent true and false. Therefore, it is important to convert between Boolean expressions and appropriate integer values. An array, structure, or union cannot be represented with a single Yices variable. The Yices language does not support pointers; therefore, we must ensure that any operations involving pointers are handled safely. Furthermore,

Figure 16: Example code and the corresponding Yices formulas

C Code	Yices Formulas for Path 1	Yices Formulas for Path 2
<pre> int radius, compute; float result;  radius = 2;  compute = AREA;  if (compute == AREA)      result = 3.14 * radius * radius;  else      result = 2 * 3.14 * radius; </pre>	<pre> ;; variable declarations (define radius_1::int) (define compute_1::int) (define result_1::real)  ;; variable assignments (define radius_2::int 2) (define compute_2::int compute_1) (define result_2::real result_1) (define radius_3::int radius_2) (define compute_3::int 1) (define result_3::real result_2)  ;; an assertion (assert (/= (ite (= compute_3 1)                 1 0) ) 0)  ;; variable assignments (define radius_4::int radius_3) (define compute_4::int compute_3) (define result_4::real   (* (* 3.14 radius_3) radius_3)) </pre>	<pre> ;; variable declarations (define radius_1::int) (define compute_1::int) (define result_1::real)  ;; variable assignments (define radius_2::int 2) (define compute_2::int compute_1) (define result_2::real result_1) (define radius_3::int radius_2) (define compute_3::int 1) (define result_3::real result_2)  ;; an assertion (assert (= (ite (= compute_3 1)                1 0) ) 0)  ;; variable assignments (define radius_4::int radius_3) (define compute_4::int compute_3) (define result_4::real   (* (* 3.14 2) radius_3)) </pre>

appropriate Yices formulas must be written out for SCCs and the CFG nodes reachable from the SCCs to correctly keep track of the value of each variable.

The following subsections discuss how to handle the different types that can be used in a C program: scalar types, Booleans, arrays, structures and unions, and pointers, as well as how Yices formulas are written to safely approximate the effects of loops and recursions.

### 3.4.1 Scalar Variables

It is a straightforward task to write Yices formulas for scalar variables. As discussed above, a Yices variable name consists of the corresponding C variable name and a subscript that corresponds to the current assignment. There are two primitive Yices types, `int` and `real`, for scalar variables into which we need to map all of the scalar types in C. This process is trivial since there are only integer and floating-point primitive types in C which correspond to `int` and `real` types in Yices. The right-hand side of an assignment can be a literal, a previously defined Yices variable, or a function of these elements. This is illustrated in the code previously shown in Figure 16.

### 3.4.2 Boolean Expressions

The C language does not evaluate logical and relational expressions into Boolean values. Rather, it evaluates them into integers, either 0 or 1 which implicitly indicates false or true, respectively. In contrast, many other languages including Yices explicitly evaluate such expressions into Boolean values, either true or false. Therefore, we must take an additional step to convert between the Yices Booleans and the C integers to accommodate this particular characteristic of the C language.

First, we must ensure that the Yices Boolean values are converted into the C integers. For instance, the C code shown in Figure 17 assigns the evaluated integer value of  $(x == y)$  to the variable `w` at line 1. In the Yices formulas (a simplified version that shows only the formula for the killed variable at each assignment), this expression is evaluated as a Boolean whereas, in C, it is evaluated as an integer. This problem is solved by using the Yices *if-then-else* operation `ite`, to convert true and false into 1 and 0, respectively. This helps us to avoid type mismatches between Boolean values and integer values.

Second, we must handle conditional expressions such as logical or relational expressions in an `if` statement which involves converting the C integers into the Yices Booleans. In the C language, such an expression is implicitly compared to 0 and evaluated as false if it is equal to 0 or true otherwise. Consider the C code shown below in Figure 17 which has a relational expression in the `if` statement at line 2. There are two assertions associated with this statement since there are two paths, one with the *then* branch and another one with the *else* branch. The relational expression

Figure 17: Example code that uses a relational expression in an assignment

C Code	Yices Formulas for Path 1 (simplified)	Yices Formulas for Path 2 (simplified)
<pre> 1: w = (x == y); 2: if (x == y) 3:   w = 1; 4: else 5:   w = 0; </pre>	<pre> ;; assume current subscript # is 2 (define w_2::int (ite (= x_1 y_1) 1 0)) (assert (/= (ite (= x_2 y_2) 1 0) 0)) ;; more formulas here </pre>	<pre> (define w_2::int (ite (= x_1 y_1) 1 0))  (assert (= (ite (= x_2 y_2) 1 0) 0)) ;; more formulas here </pre>

itself is converted to 0 or 1 using the Yices if-then-else operation. Then, each assertion compares the converted value of the expression to 0 for equality or inequality, based on which branch is taken in that path, as illustrated in Figure 17. (Note that this is how control-points in C actually work.)

### 3.4.3 Arrays

The Yices language does not support array types, but there are several ways to handle arrays in Yices formulas. It is more complicated to write Yices formulas for an array than for a scalar variable because we must consider every element in the array rather than just the array itself. For example, a statement  $A[i] = 0$ , where the array-index  $i$  is unknown, tells us that some element in  $A$  is assigned 0. Since  $i$  is unknown we must ensure that all elements in  $A$  are unknown. There are two broad approaches to writing Yices formulas that correctly handle arrays:

1. Treat each element of an array as an individual Yices variable.
2. Use Yices function types.

The first approach is undesirable because we may end up writing too many formulas for all the array elements used in the program. Moreover, if an array index is not an integer literal, we cannot determine the target Yices variable (which also requires writing Yices define operations for *all* elements of the array to define them to be unknown). The second approach considers an array as a function that takes one argument: the array index, and returns the value of the corresponding element. This works fine in Yices for the purpose of simulating arrays although C does not treat arrays that way.

For multi-dimensional arrays, the function of the current rank will return another function that corresponds to the next rank. The function that corresponds to the highest rank returns the value of the element. For example, let  $A$  be an array of size  $2 \times 2$  and of type integer. In C, there would be a contiguous block of memory allocated for the array, as shown in Figure 18a. However, in Yices formulas, it is better understood as an array of arrays where the function corresponding to the first rank returns another function that corresponds to the second rank, as illustrated in Figure 18b.

Figure 18: Array of size  $2 \times 2$  in the C language and in Yices formulas

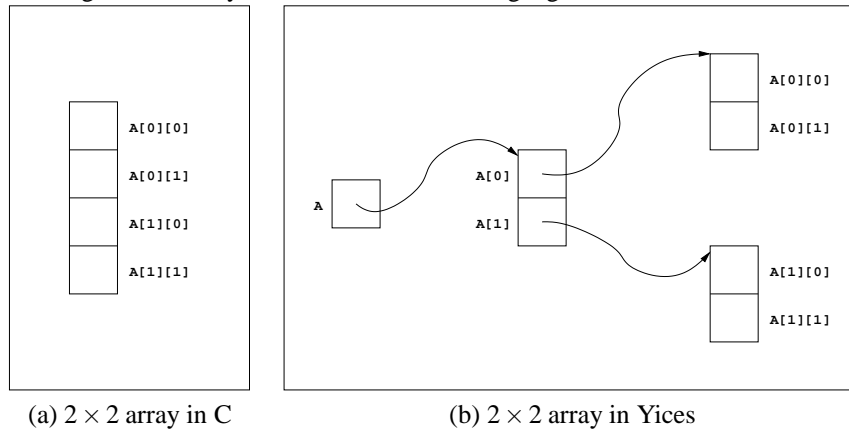


Figure 19: Example code that uses arrays and corresponding Yices formulas

C code	Yices Formulas (simplified)
1: int A[2];	(define A_1::(-> int int)) ;; declaration
2: int B[2][2];	(define B_1::(-> int (-> int int))) ;; declaration
3: A[0] = 1;	(define A_2::(-> int int) (update A_1 0 1))
4: B[0][1] = 2;	(define B_3::(-> int (-> int int)) (update B_2 0 (update (B_2 0) 1 2)))

Consider the example in Figure 19 which shows C code that uses arrays and the corresponding (simplified) Yices formulas. Each time an array or an array element is defined, a new instance of the array is defined as a function that takes an array-index argument and returns the appropriate element (which could be another array or just an element). For each update of an array element, we define the next instance of the array as a new Yices variable and assign the “updated” version of the previous instance of the array with a new value for the element. For example, Yices variable  $A\_2$  is defined and assigned an “updated” version of  $A\_1$  where the 0th element gets a new value, 1, for the corresponding C assignment at line 3 in Figure 19.

### 3.4.4 Structures and Unions

In the C language, structures are used to define new data types by clustering a group of variables. Unions are similar to structures except that all members share the same memory address. Since both structures and unions are frequently used in practice, our conditioned slicing tool must handle these types properly to be suitable for real C programs. The code shown below in Figure 20 is used to illustrate how we handle structures.

There are two approaches to writing Yices formulas for structures:

1. Treat each scalar field as an individual Yices variable.

Figure 20: Example code that uses structures

```
1: struct V {int x; int y};
2:
3: struct V incAll(struct V arg) {
4:     arg.x = arg.x + 1;
5:     arg.y = arg.y + 1;
6:     return arg;
7: }
8:
9: void main() {
10:     struct V t1, t2;
11:     t1.x = 1;
12:     t1.y = t1.x;
13:     t2 = t1;
14:     t1 = incAll(t1);
15: }
```

## 2. Use Yices record types.

In most cases, CodeSurfer provides a unique identifier for each field of each structure, and it normalizes each whole-structure assignment into a sequence of field assignments. For instance, the whole-structure assignment  $t2 = t1$  at line 13 in Figure 20 is normalized into two field assignments:  $t2.x = t1.x$  and  $t2.y = t1.y$ . For these cases, the first approach (treating each field as an individual Yices variable) seems most natural.

However, CodeSurfer does not provide unique identifiers for the fields of the temporary variables used to pass structure parameters or to return structure results, and it does not normalize the structure assignments involved in passing/returning structures. In the PDG for the example, the argument  $t1$  is passed to function `incAll` via two PDG nodes that represent the whole-structure assignments  $\$param1 = t1$  and  $arg = \$param1$ . Similarly, the final value of  $arg$  is returned via PDG nodes that represent the whole-structure assignments  $incAll\$result = arg$  and  $t1 = incAll\$result$ . (The problem still remains although we simplified these as  $arg = t1$  and  $t1 = arg$  for convenience.)

Because of the way CodeSurfer handles parameters and function results, treating each field of a structure as an individual Yices variable would require generating unique identifiers for the fields of structure parameters and return values. This can become complicated, especially when we have arrays of structures whose fields are arrays, etc. Therefore, we have chosen to use Yices record types for structures.

Our approach to handling structures involves the following steps:

- Define a Yices record type for each structure type defined in the code (`struct V` in the example).
- Use the appropriate record type to define an initial Yices variable for each structure variable declared in the code

Figure 21: Yices formulas for the code in Figure 20

Normalized C Code	Corresponding Yices Formulas (simplified)
<code>struct V {int x; int y};</code>	<code>;; define record type (define-type V (record x::int y::int))</code>
<code>struct V arg;</code>	<code>;; initial variable declarations (define arg_1::V)</code>
<code>struct V t1;</code>	<code>(define t1_1::V)</code>
<code>struct V t2;</code>	<code>(define t2_1::V)</code>
<code>t1.x = 1;</code>	<code>;; new declarations for structure updates (define t1_2::V (update t1_1 x 1))</code>
<code>t1.y = t1.x;</code>	<code>(define t1_3::V (update t1_2 y (select t1_2 x)))</code>
<code>t2.x = t1.x;</code>	<code>(define t2_4::V (update t2_3 x (select t1_3 x)))</code>
<code>t2.y = t1.y;</code>	<code>(define t2_5::V (update t2_4 y (select t1_4 y)))</code>
<code>arg = t1 // call</code>	<code>(define arg_6::V t1)</code>
<code>arg.x = arg.x + 1</code>	<code>(define arg_7::V (update arg_6 x (+ (select arg_6 x) 1)))</code>
<code>arg.y = arg.y + 1</code>	<code>(define arg_8::V (update arg_7 y (+ (select arg_7 y) 1)))</code>
<code>t1 = arg // return</code>	<code>(define t1_9::V arg_8)</code>

(variables `arg`, `t1`, and `t2` in the example).

- For each use of an individual structure field-e.g., `arg.x` and `arg.y` on lines 4 and 5, `t1.x` on line 12 and the use of all fields of `t1` on line 13 (due to CodeSurfer’s normalization of that whole-structure assignment)-use a Yices *select* operation to get the value of that field.
- For each update of a structure variable `v`, define the next instance of `v` as a new Yices variable. A whole-structure update of `v` (for a parameter or returned value) is handled just like an update to a scalar variable, while an assignment to an individual field of `v` involves using a Yices *update* so that the new instance of `v` is the same as the previous one except at the field that was assigned to.

Our approach is illustrated in Figure 21, which shows the (normalized) code from Figure 20, with the corresponding Yices formulas.

Unions are very much like structures except that an assignment to a field overwrites the values of all other fields in the union variable. The results are implementation-dependent if a value is stored as one type and extracted as another [4]. Our tool handles this problem by assigning unknown values to all the fields with different types than the one that has just been assigned a value and assigning that same value to all fields of the same type. The unknown values are generated using special Yices variables because the Yices *update* operator requires an explicit value to be assigned. Figure 22 shows a program that uses unions, and the corresponding Yices formulas.

Figure 22: Program with unions and corresponding Yices formulas

C Code	Yices Formulas
union U {int f1; float f2};	<code>;; define record type (define-type U (record f1::int f2::real))</code>
union U u1, u2;	<code>;; initial variable declarations (define \$unknown-1::real) (define \$unknown-2::int) (define u1_1::U) (define u2_1::U)</code>
u1.f1 = 1;	<code>;; new declarations for union updates (define u1_2::U (update u1_1 f1 1))</code>
u1.f2 = 1.0;	<code>(define u1_3::U (update u1_2 f2 \$unknown-1)) (define u1_4::U (update u1_3 f2 (/ 10 10))) (define u1_5::U (update u1_4 f1 \$unknown-2))</code>
u2 = u1;	<code>(define u2_6::U (update u2_5 f1 (select u1_5 f1))) (define u2_7::U (update u2_6 f2 (select u1_6 f2)))</code>

### 3.4.5 Pointers

It is difficult to represent C pointers in Yices because the Yices language does not support reference data types. (Pointers can be represented as integers since they represent memory addresses. The real problem is that we cannot refer to a Yices variable.) Therefore, we took a simple step and disregarded all the variables that are *pointed-to* by some pointer(s) when writing the Yices formulas. The *pointed-to-by-set* of a variable, which is computed by CodeSurfer during the project build, is a set of pointers that may point to that variable at some point of the program (flow-insensitive).

If a variable has a non-empty pointed-to-by-set, we remove that variable from the set of variables for which the Yices formulas are written. When the variable is used in the right-hand side of an assignment, it is evaluated to a unknown value using a special Yices variable. This is done for pointer dereferences, too. When a pointer dereference occurs in the left-hand side, we do not write any formulas since formulas for the variables that may be pointed-to-by pointers are not written anyway. A pointer dereference that occurs in the right-hand side is evaluated to a unknown value since we cannot determine the exact variable that the pointer points to at that moment due to the limitation of the pointer analysis.

Yices formulas for pointers can still be written out as assignments to the corresponding Yices variables (of type integer). Address-of expressions are handled as if they are evaluated to non-zero values since no variable can be allocated at address 0. Note that this helps us at least to determine whether or not a pointer is null despite the oversimplification. The (simplified) Yices formulas in Figure 23 illustrates how pointers are handled. For address-of expressions, we use

Figure 23: Program with pointers and corresponding Yices formulas

C Code	Yices Formulas
<pre> /* Ptd-to-by(v) = {p}, Ptd-to-by(v) = {p} */ 1: int v, w, x; 2: int *p;  3: v = 0; 4: w = 0; 5: p = &amp;v; 6: *p = 1 7: p = &amp;w; 8: x = *p + v; 9: p = NULL; </pre>	<pre> ;; no formulas for v, w because they are pointed to by p (define x_1::int) (define p_1::int) (define \$unknown-1::int) (define \$unknown-2::int)  ;; no formulas for v because it is pointed to by p ;; no formulas for w because it is pointed to by p (define p_2::int (subtype n::int (/= 0))) ;; no formulas for lhs pointer dereference (define p_3::int (subtype n::int (/= 0))) (define x_4::int (+ \$unknown-1 \$unknown-2)) (define p_5::int 0) </pre>

the *subtype* operator to restrict the associated variable to hold a non-zero value.

### 3.4.6 Strongly-connected Components

The purpose of writing Yices formulas for each assignment in a path is to correctly approximate the program state at that point in the path. Thus, we must ensure that appropriate formulas are written for the SCCs and CFG nodes reachable from the SCCs that occur in the paths that we generate. The simplest approximation that can be used for an SCC is assigning unknown values to all the variables that may be killed in the SCC and the CFG nodes reachable from that SCC. For example, the CFG in Figure 13 has two variables that may be killed in the SCC and the CFG nodes reachable from that SCC, *i* and *g*. Thus, we write two define operations to assign unknown values to both *i* and *g*.

## 4 Future Work

This section describes possible improvements to the implementation for the future. As discussed earlier, there are two major areas that we have left overly simplified: strongly-connected components (SCCs) and pointers. Section 4.1 explains how we can handle SCCs better by writing more fine-grained formulas for the SCCs. Section 4.2 describes how we can exploit some of the pointer analyses to reduce the size of a slice.

### 4.1 Strongly-connected Components

Previously, we introduced the concept of “collapsing” the CFG nodes in an SCC into a list of CFG nodes to represent an SCC and functions called from that SCC. This simplifies the Yices formulas a bit too much because most C programs

Figure 24: Example code used to illustrate how better treatment of loops can be used to reduce the size of a slice

Original Program	Conditioned Slice 1	Conditioned Slice 2
<pre> 1: void main() { 2:   int a, b, x; 3:   x = 0; 4:   scanf("%d", &amp;a); 5:   scanf("%d", &amp;b); 6:   while (a &gt; 0 &amp;&amp; b &lt; 10) { 7:     x = x + a + b; 8:     scanf("%d", &amp;a); 9:     scanf("%d", &amp;b); 10:  } 11:  x = x * 2; 12:  if (b &lt; 10) 13:    x = x * 2; 14:  else 15:    x = x * 4; 16:  printf("x is %d", x); 17: }</pre>	<pre> 1: void main() { 2:   int a, b, x; 3:   x = 0; 4:   scanf("%d", &amp;a); 5:   scanf("%d", &amp;b); 6:   while (a &gt; 0 &amp;&amp; b &lt; 10) { 10:  } 11:  x = x * 2; 12:  if (b &lt; 10) 13:    x = x * 2; 14:  else 15:    x = x * 4; 16:  printf("x is %d", x); 17: }</pre>	<pre> 1: void main() { 2:   int a, b, x; 3:   x = 0; 4:   scanf("%d", &amp;a); 5:   scanf("%d", &amp;b); 6:   while (a &gt; 0 &amp;&amp; b &lt; 10) { 7:     x = x + a + b; 8:     scanf("%d", &amp;a); 9:     scanf("%d", &amp;b); 10:  } 11:  x = x * 2; 12:  if (b &lt; 10) 14:  else 15:    x = x * 4; 16:  printf("x is %d", x); 17: }</pre>
(a)	(b)	(c)

use loops and recursion extensively. Furthermore, this oversimplification may result in considering too many paths as feasible. This is because the “collapsing” technique does not take into account useful information about the behavior of the cycle such as the condition under which a loop terminates. We believe that this can be improved by writing out the Yices formulas for SCCs at a more fine-grained level.

The code shown in Figure 24a is used to illustrate the possible improvements. Let us assume that the POI is the `printf` statement at line 16. The “normal” slice would include the whole program because all of the PDG nodes are reachable going back from the PDG node corresponding to the POI.

If we also assume that the user-supplied condition is  $(a == 0)$  at line 5, then we know that the loop never executes, because the condition is not satisfied at the zeroth iteration; i.e., any path that follows the true branch out of the loop condition is infeasible. Therefore, we can remove lines 7 through 9 (the loop body) from the PDG before slicing, as shown in Figure 24b.

If instead we assume that the user-supplied condition is  $(a == 10)$  at line 11, then we do not know anything about how many times the loop executed, but we do know that the loop exited because `b` was *not* less than 10. Therefore, we know that the *then* branch of the `if` statement on line 12 is only on infeasible paths, which allows us to eliminate line 13, as shown in Figure 24c.

Our suggested improvement to write out Yices formulas for loop SCCs involves writing out the paths associated with zero iterations of the loop, and writing (an approximation to) the paths associated with one or more iterations.

In the first case, we simply write an assertion of the negation of the loop condition (and ignore the loop body). In the second case, we assert that the loop condition is true, we define all variables that may be killed in the loop to be unknown, and then assert the negation of the loop condition.

## 4.2 Pointers

Recall that we disregarded all the variables that may be pointed-to by some pointer(s) when writing Yices formulas because we cannot determine which variables may be pointed-to by a particular pointer at each assignment. Using the *points-to-sets* (the sets of variables that may be pointed-to by each pointer, which are computed by CodeSurfer during the project build), however, we can write better Yices formulas. In particular, we can write formulas for pointed-to variables that are assigned to directly, rather than through a pointer dereference. To understand how this can improve on our current technique for handling pointers, consider the example code in Figure 25a.

Let us assume that the POI is the `printf` statement at line 16 and the user-supplied condition is `(a == 0)` at line 6. Using the current implementation, the conditioned slice includes every line of the code because all paths are considered feasible. All paths are considered feasible because no definitions are written for the assignments to variables `x` or `y` because they may be pointed-to by `p`, and therefore Yices cannot tell whether the conditions of the `if` statements are consistent with the user-supplied condition or not.

If we use the *points-to-sets*, however, we can reduce the size of the slice by writing a definition for variable `x` at line 6, giving it the current value of variable `a`. In this case, since the user-supplied condition says that `a` is zero, Yices can infer that `x` is also zero, which means that the condition of the `if` statement at line 7 must be true, so all paths that take the *else* branch are infeasible. This allows us to eliminate lines 9 and 10 from the slice, as shown in Figure 25b.

This approach still fails to determine that the *then* branch of the `if` statement at line 12 is only on infeasible paths (because the assignment via the dereference of `p` at line 11 would cause both `x` and `y` to be given unknown values, since `p`'s *points-to-set* includes both `x` and `y`). To overcome this limitation, we can take a step further and keep track of which variable each pointer points to after each pointer assignment when writing out formulas for a path. In the example, the path that goes through the *then* branch of the first `if` statement causes `p` to point to `y`. Therefore, the formulas written for the assignment `*p = *p + 1` at line 11 would set `y = 1`, and would leave `x` unchanged. This in turn would allow Yices to determine that the path that takes the *then* branch of the first `if` statement and the *else* branch of the second `if` statement is infeasible. This allows us to eliminate lines 9, 10, and 13, as shown in Figure 25c.

Figure 25: Example code used to illustrate how better treatment of pointers can be used to reduce the size of a slice

Original Program	Conditioned Slice 1	Conditioned Slice 2
<pre> 1: void main() { 2:   int x, y, a; 3:   int *p; 4:   y = 0; 5:   scanf("%d", &amp;a); 6:   x = a; 7:   if (x == 0) 8:     p = &amp;y; 9:   else 10:    p = &amp;x; 11:  *p = *p + 1; 12:  if (y &gt; 10) 13:    a = 20 + *p; 14:  else 15:    a = 30 + *p; 16:  printf("a is %d", a); 17: }</pre>	<pre> 1: void main() { 2:   int x, y, a; 3:   int *p; 4:   y = 0; 5:   scanf("%d", &amp;a); 6:   x = a; 7:   if (x == 0) 8:     p = &amp;y; 11:  *p = *p + 1; 12:  if (y &gt; 10) 13:    a = 20 + *p; 14:  else 15:    a = 30 + *p; 16:  printf("a is %d", a); 17: }</pre>	<pre> 1: void main() { 2:   int x, y, a; 3:   int *p; 4:   y = 0; 5:   scanf("%d", &amp;a); 6:   x = a; 7:   if (x == 0) 8:     p = &amp;y; 11:  *p = *p + 1; 12:  if (y &gt; 10) 14:  else 15:    a = 30 + *p; 16:  printf("a is %d", a); 17: }</pre>
(a)	(b)	(c)

## 5 Conclusion

This thesis presented an improved conditioned slicing technique, our implementation of the tool that works for C programs, and possible improvements that can be done in the future. The conditioned slicing technique [3] is improved by taking into account data dependences that are only induced by infeasible paths. We implemented the tool for C programs using CodeSurfer and the Yices theorem prover. There have been difficulties and limitations during the implementation of the tool due to the time constraint and the capabilities of the theorem prover. Future work includes implementing suggested improvements to make the tool more suitable for real C programs as well as empirical evaluation of conditioned slicing by comparing the slices produced by our conditioned slicing tool and by the “normal” slicing tool.

## Acknowledgements

I would like to thank my thesis advisor, Professor Susan Horwitz, who introduced me to the topic and guided me throughout the writing of this thesis. My thanks also go to Dan Bull for being an “esteemed” colleague, Wooseok Chang for being a spritual and academic mentor, and Professor Charles Fischer and Professor Benjamin Liblit for teaching me in their unforgettable classes. Last, but not least, I thank my parents, whom I love most, for their endless love and support.

## References

- [1] K.J. Ottenstein and L.M. Ottenstein. The program dependency graph in a software development environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment*, 1984.
- [2] M. Weiser. Program slicing. *IEEE Transaction on Software Engineering*, 1984.
- [3] C. Fox, M. Harman, R. Hierons, and S. Danicic. Backward conditioning: a new program specialisation technique and its application to program comprehension. *9th International Workshop on Program Comprehension. IEEE Computer Society*, 2001.
- [4] B. Kernighan and D. Ritchie. *The C programming language. Prentice Hall*, 1988.
- [5] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 1981.
- [6] Grammatech. CodeSurfer. <http://www.grammatech.com/products/codesurfer/overview.html>
- [7] Computer Science Laboratory of SRI International. Yices: an SMT solver. <http://yices.csl.sri.com/>