Journal of Information Technology and Engineering
Vol. 1 No. 2 (December, 2016)

Predicate Abstraction of C Programs

V. K. Pachghare[1]

[1]Lecturer, Computer Engineering & IT Department, College of Engineering, Pune
(An Autonomous Institute of Government of Maharashtra, India) *E-mail: vkp@comp.coep.org.in*

Received: 03rd March 2016 Revised: 14th July 2016 Accepted: 20th August 2016

Abstract: *Typically, software contains millions of lines of code which presents a major challenge for program analysis and related techniques such as model checking. Analyzing such large code is expensive in terms of space and time. Predicate abstraction is a technique that generates an abstract of the original program using a set of predicates. Such an abstract or reduced code preserves all the properties of the original program. It is achieved through a technique called program slicing which eliminates all the irrelevant parts of the original program with respect to the set of predicates. Moreover, the abstract code can be used as an input to a model checker to prove properties of programs. This is vital since a model checker accepts only a finite state system.*

This paper covers concepts related to predicate abstraction and our approach to building the predicate abstraction tool for C programs. It also illustrates how this tool is unique as compared to other tools in the software industry. The important applications of the tool along with the challenges faced are covered in brief as well.

Keywords: *Predicate Abstraction, Program Slicing, Approach, Challenges*

1. INTRODUCTION

Program analysis refers to the process of collecting information about a program. Program analysis is a key component of many applications and software development processes. It is required for applications such as compilers / decompilers, reduces costs of program maintenance and increases effectiveness of program testing.A growing commercial use of program analysis is in the verification of properties of software used in safety-critical computer systems and locating potentially vulnerable code.

Model checking is a technique related to program analysis which is used to prove properties of finite state systems. However, generally programs contain thousands of lines of code which results in infinite state systems. Such infinite state systems cannot be given as input to the model checker (model checking tool) due to memory restrictions. Thus there arises a need to abstract the original program with respect to a set of predicates to generate a finite state system that can fit into the model checker. The generated abstraction is such that it preserves the behaviour of the original program. Such an abstraction is called Predicate Abstraction. There are many important applications of predicate abstraction viz. deadlock detection, divide by zero detection, etc.

2. PREDICATE ABSTRACTION

Predicate abstraction is a technique that is used to prove properties of infinite state systems. It is a combination of theorem proving and model checking techniques. Given a concrete infinite state system and a set of abstraction predicates (points of interest in a particular program, expressed as variables), a conservative finite state abstraction is generated. Predicate abstraction is conservative in the sense that for every execution in the concrete system there is a corresponding execution in the abstract system.

Abstraction may be very coarse .So we need to refine it. However, abstraction is expensive. Moreover the reachable state space of a program is generally sparse. Hence we use refinement only where required by constructing the abstraction 'on the fly' driven by the search.

This gave rise to Lazy abstraction. Lazy abstraction continuously builds and refines a single abstract model on demand just enough to verify the desired property. When applying iterative abstraction refinement, we must strike a delicate balance between the accuracy of the abstraction that is produced and the speed at which it is generated.

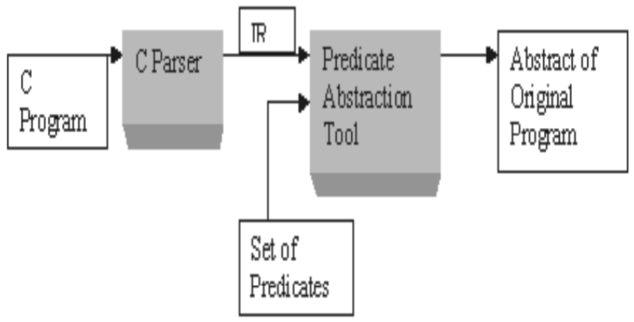


Figure 1: Components in C Programs

The block diagram shown in figure 1 shows the components involved in predicate abstraction of C programs.

1. C Parser: It takes a C program as its input and generates an intermediate representation (IR).
2. Predicate Abstraction Tool: This is responsible for analyzing the whole program and generates an abstract version of the program. It takes the intermediate representation of the C program and a set of predicates which generates the abstract of the program as its output.

Predicate abstraction involves a technique called program slicing.

3. PROGRAM SLICING

Program slicing is a technique for simplifying programs by focusing on selected aspects of semantics. The process of slicing deletes those parts of the program which can be determined to have no effect upon the semantics of interest. Figure 2 gives the programmers view of a slice.

Slicing could be of various types:

1. Static: identifies all program code that can in any way affect the value of a given variable
2. Dynamic: Dynamic information(input to the C program) is given to compute the slice along with the information to compute the static slice
3. Conditioned: Slice computed with respect to certain conditions whose result is known.

Our paper will be dealing with static slicing.

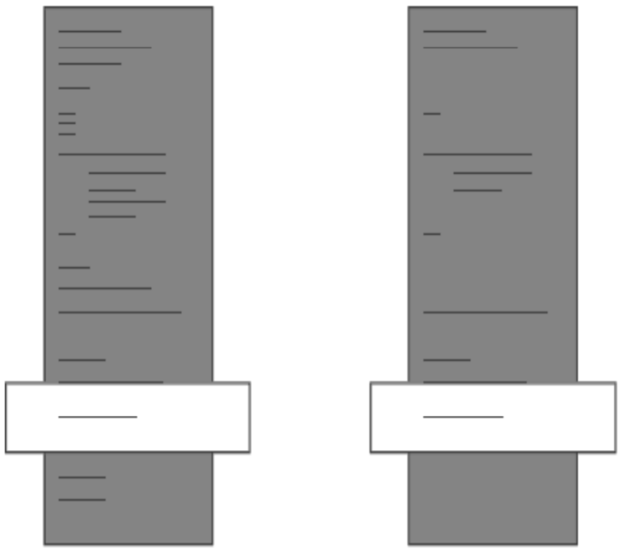


Figure 2: Programmer's View of a Slice

3.1 Static Slicing

There are many forms of slice, so it will be helpful to start off with a simple form of slice; the static slice. The other forms of slice can be thought of as augmentations of this static form.

A slice is constructed by deleting those parts of the program that are irrelevant to the values stored in the chosen set of variables at the chosen point. The point of interest is usually identified by annotating the program with line numbers which identify each primitive statement and each branch node.

The point of interest will be indicated by adding a comment to the program. In general slices are constructed for a set of variables, but in this article only slice constructed for a single variable will be considered. Thus, given a variable *v* and a point of interest *n*, a slice will be constructed for *v* at *n*. This is not restrictive, because the slice with respect to a set of variables *V* can be formed from the union of the slices on each variable in V.

Simple example of static slicing is given below:

```
Void main()                   Void main()
{                              {
   int I,j,k;                    int I,j;
   j=2;                          j=2;
   i=j;                          i=j;
   k=3;                          printf("%d",i):
  printf("%d",i);              }
  printf("%d",k);
}
A:Original program            B:Sliced Program

Slicing Criteria: Variable 'i'
```

Explanation: In the above example, A is the original program and B is the corresponding sliced program with the slicing criteria as variable 'i'. As it can be clearly seen the assignment k=3 and the print f("%d",k) bears no relevance to the value of 'i'. Hence, these statements are deleted to obtain the sliced program, B.

4. WORK DONE IN THIS FIELD

Microsoft has been doing a lot of research in this field. SLAM project is an example of the work they have done. The aim of the SLAM project was to prove correctness properties of OS device drivers by using predicate abstraction.

Berkeley University has made significant contributions to this field. BLAST (Berkeley Lazy Abstraction Software Verification Tool) constructs an abstraction of the original system 'on the fly' only according to the reachable states in the program.

MAGIC (Modular analysis of programs in C) follows the counterexample guided abstraction refinement paradigm.

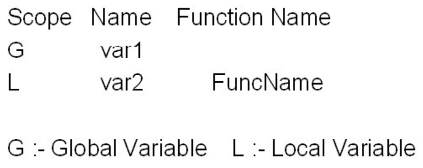
All these approaches combined abstraction along with the model checking tools to prove program properties.

5. OUR OBJECTIVE AND APPROACH

Work that has been done in this area has combined slicing along with model checking. Our objective here is to create a pure slicing tool that will mainly be used for scaling up data flow analysis. Scaling up will be done with respect to memory as well as time. Hence, very large programs can be analysed in less time.

We have taken a rather naïve approach to slicing. As mentioned above, abstraction is expensive and hence initially the abstraction is coarse. It can be refined as per the need.

We are following a user-driven approach in which we give a set of predicates to our program from a file. These predicates form our points of interest in the program. With respect to these predicates the predicate abstraction prototype tool will give all the parts of the program that affect the value of these predicates / program variables. In other words, it gives us the parts of the program relevant to the set of predicates input to the program. The input set of predicates is given to the program in the form:



```
Scope   Name    Function Name
G        var1
L        var2        FuncName

G :- Global Variable    L :- Local Variable
```

Figure 3: Input File Format

Here 'G' stands for a global variable and 'L' stands for a local variable. The scope of the variable is given followed by the name of the variable which is followed by the function to which it belongs.

In the example above we can see that the scope is 'G' or global and the name of the variable is 'var1'. Being global it does not belong to any function. The second variable has a local scope, name as 'var2' and belongs to 'FuncName' function.

In our tool we are interested in the 'Divide by Zero' property. For testing purposes a driver has been made that generates the predicates from a C application. In this case it will consider all the divisors in the program and collect all the variables in the divisors as the points of interest. It will generate a file in the form given in figure 3.

Once this input has been given to our tool, it returns all the relevant aspects of the program in the form of functions relevant over the set of input predicates. The approach to find such relevant functions is as follows:

Initially there are three sets computed for each function which are as follows:

1. Globals used
2. Globals modified
3. Formal parameters modified

The first set gives all the global variables that have been used by a particular function and all its children (the functions that are called in its body) since they may be using global variables.

The second set gives all the global variables that have been modified by a particular function and all its children.

The third set gives us all the formal parameters that are modified by a particular function and its children. It means that after a function has been executed the memory locations that it has modified are contained in this set in the form of formal parameter numbers since it may be called from any function. Once these sets have been computed we proceed towards the main approach.

Consider the functions in a program one at a time.

First of all check if our point of interest variable is being directly modified in this function. If yes, then mark this function as relevant and mark all its ancestors as relevant. The ideology behind this is that the value of our point of interest can be modified only if the parent of our function is called. So, only if the parent is called will the current function be called.

The next case to be considered is during the time of function calls inside a function definition. It is because a call to a function could change a memory location which in turn could affect the values of our points of interest. A call could change a memory location by retuning a value or by modifying a location pointed to by the pointer sent as a formal parameter or by changing global variables. Hence, when there is a function call we consider three cases.

1. It call could be a call that returns a value and the value is being used (the value returned is being assigned to something)
2. It could take a reference as its parameter
3. It could just be a plain call that does not return a value or it returns a value but the value is not used or no reference is being passed to it.

Case 1: Consider the first case. If the call returns a value and the value is being used (the value returned is modifying some memory location), then we may assume that the value being returned may later affect the values of our points of interest. Hence we mark the called function and all its ancestors (all its callers, and their callers and so on) as relevant since this function can be called only if its caller is called. We also need to make an executable program. Hence the above rule.

Case 2: In the second case since a reference is being passed as a parameter to the function we can find whether this function is modifying some memory location by checking the third set that we had initially computed. If yes, then we may assume that the modification may later affect the values of our points of interest. Hence we mark the called function as relevant and all its ancestors as relevant. If no, then we need to check for the global variables that the function modifies in the second set that has been computed. If we find that the called function is modifying some global value which may be used by the caller function or any of its ancestors we mark the called function as relevant and all its ancestors as relevant.

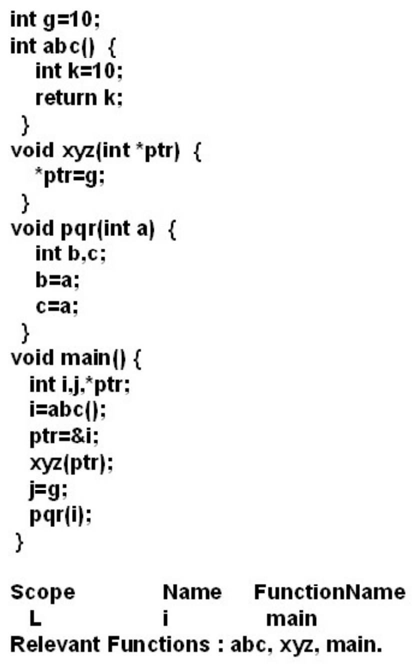
Case 3: In the third case, we bring into picture global variables. If we find that the called function is modifying some global value which may be used by the caller function or any of its ancestors we mark the called function as relevant and all its ancestors as relevant. It is done so because we assume that the change in the global variables may later affect the points of our interest.

In this way we get all the relevant parts of the program, removing all the irrelevant parts of the original program.

Let us illustrate with an example

In the following example variable ‘i’, a local variable that belongs to function ‘main’ is our point of interest. According to our approach we have found that the relevant functions are ‘abc’, ‘xyz’ and ‘main’. Function ‘abc’ is relevant because it is returning a value inside the function ‘main’ that is being consumed. Hence ‘abc’ is relevant. Also ‘main’ is relevant because the value of of ‘i’ is being directly modified inside this function. Function ‘xyz’ is relevant because a pointer is being passed which according to our second case should be marked as relevant. Since function ‘pqr’ does not fall into any of the above cases it is irrelevant with respect to variable ‘i’.

In this manner we can find out all the functions that may directly or indirectly affect the value of our point of interest.



```
int g=10;
int abc() {
   int k=10;
   return k;
 }
void xyz(int *ptr) {
   *ptr=g;
 }
void pqr(int a) {
   int b,c;
   b=a;
   c=a;
 }
void main() {
  int i,j,*ptr;
  i=abc();
  ptr=&i;
  xyz(ptr);
  j=g;
  pqr(i);
 }

Scope        Name    FunctionName
  L            i        main
Relevant Functions : abc, xyz, main.
```

6. CHALLENGES IN PREDICATE ABSTRACTION

In Predicate Abstraction of C programs various aspects of C Language are to be considered, so various challenges were encountered in designing the Predicate Abstraction Tool. Some of the challenges faced are as follows:

- **Pointers**: Pointers are datatypes which hold addresses of variables to which it is ‘pointing to’. In C programs pointers directly affect the value of variable to which it is referencing. Thus in Predicate Abstraction it is necessary to keep track of all the pointers which directly or indirectly may affect our point of interest variable. Since we process functions not according to the flow of the program it is even more difficult to handle pointers. When a pointer is passed as parameter to a call, it is important to note whether the pointer is modifying a memory location after retuning from the call.
- Procedures or Functions : In Predicate Abstraction the Procedures or Functions which are modifying the value of point of interest variable are relevant to us, while others are irrelevant and are sliced out in the abstract version of a program. There can be various types of functions :
  1. Call by reference.
  2. Call by value.
  3. Call without parameters etc.

Thus all the functions in a program are analyzed and corresponding operations are carried out to find all the functions which are relevant to our point of interest variable.

Recursive Calls: All the recursive functions in a program are considered to find if it is relevant or irrelevant. The procedure to find relevant functions is as follows:

a. Process

b. Mark

By means of a flag we can mark the function as processed in the beginning itself. In this way we can handle recursive functions.

- Complex Data Types: Structures, unions and arrays are handled differently. For instance, even if a member of a structure variable is being modified we need to consider the structure variable as modified. Doing this requires an API called coveredby( ) which tells whether the variable under consideration is part of the structure variable.
  e.g.:

```
                    Struct node
                    {
                              Int x;
                              Char y;
                    }
Struct node s;
s.x=30;
```

Consider the structure variable‘s’ to be our point of interest. We can see that‘s’ is modified in the statement s.x=30. Using covered by API we can find whether s.x is a part of s. Hence in this manner we can handle complex data structures as well.

7. UNIQUENESS

The slicing tool differs from the ones given above. Its features are given below in brief.

Table 1
Characteristics of the Tool

Characteristics	Our Tool
Abstraction level	Function
Type of result	Set of functions
Intended application	Program analysis
Slicing Direction	Backward
Type of information	Static
The statement	Not at a particular statement

Thus the effective output from the slicing tool is a set of functions relevant to the slicing criterion, the criterion being a set of program variables.

8. APPLICATIONS

The predicate abstraction tool can be used for various applications. Some of them are given below.

- Deadlock detection: In this case we can consider semaphores as the points of interest variables. With respect to these semaphores we get only the relevant parts of the code.
- Division by zero: In order to locate statements in the program that could lead to a division by zero case, all the denominators are considered as the points of interest variables.
- Null Pointer Exception: Consider all the variables that could lead to a null pointer exception as points of interest. All the functions relevant to such variables are found.

General Applications

This section describes four of the applications to which program slicing has been put. The section starts with a discussion of debugging, which was the original motivation for program slicing. Since then slicing has been applied to many other problem areas. To give a flavor for the breadth of applications two additional topics are described: comprehension and maintenance. The first of these is covered in detail, while the latter two are presented in overview.

1. Debugging

It facilitates debugging and can be used to narrow the search for the fault, by including only those statements which could have caused a fault to have occurred on the particular execution of interest.

2. Comprehension

Program comprehension becomes better by obtaining slices to understand every individual case in the program leaving the rest aside.

3. Maintenance

By obtaining a slice with respect to some property of interest we can find the parts of code that will be affected due to a change in the predicate. Hence maintenance effort is reduced as well.

CONCLUSION

The tool has successfully been tested on a number of C programs including one with around 6000 lines of code containing 163 functions in it. In order to check for the divide by zero cause, the predicates input to the tool were the divisors in the program. A Driver was written to generate such predicates. It generated the predicates in less than half a minute. After slicing, 86 relevant functions were obtained. This processing took just 45 seconds. Thus, in order to check for divide by zero, only 86 functions need to be analyzed instead of the original 163 functions. This saves analysis time and memory immensely.

The predicate abstraction tool thus built contributes greatly in scaling up of Data Flow Analysis in terms of memory and time. Thus, instead of analyzing a code that spans thousands of lines, only the relevant parts need to be analyzed just by giving the point of interest to the tool. The Relevant parts of the program are expressed in terms of relevant functions. These are those functions which directly or indirectly affect the value of the variable of interest also known as the predicate. In finding these relevant functions, care has been taken to ensure that only those functions which definitely have no bearing on the value of predicate are marked as irrelevant. All cases for relevance have been checked meticulously.

The tool is currently being tested on a C project which has around 1 lakh lines of code containing 877 functions. After analysis, 27 predicates have been generated.

REFERENCES

[1] Herbert Schildt, JAVA: Complete Reference.

[2] *www.wikipedia.org*

[3] *www.google.com*

[4] verify.stanford.edu/satyaki/research/ResearchInterests.html

[5] verify.stanford.edu/satyaki/research/PredicateAbstraction.html

[6] *http://hissa.nist.gov/unravel/*

[7] linz.ac.at/Research/Projects/ProgramSlicing/ECOOP99/Demonstration/Demo/ppframe.htm

[8] Satyaki Das , "Predicate Abstraction", Stanford University.

[9] Tommy Hoffner, "Evaluation and Comparison of Program Slicing Tools".

[10] Andrea De Lucia, "Program Slicing: Methods and Applications", Faculty of Engineering, University of Sannio.

[11] Thomas Henzinger, U C Berkely, "An Overview of Program Slicing Software Verification with Blast".

[12] David Binkley, "Program Slicing",The Wisconsin Program-Slicing Tool.