Assume(), Capture(), Verify(), Establish(): A Vocabulary for Static Program Analysis

Hessamaldin Mohammadi[1,*], Wided Ghardallou[2], Elijah Brick [1] and Ali Mili[1]
[1]Ying Wu College of Computing, NJIT, Newark, NJ, USA
[2]University of Sousse, Sousse, Tunisia
hm385@njit.edu, wided.ghardallou@gmail.com, eb275@njit.edu, mili@njit.edu
*corresponding author

Abstract—We propose a set of functions that a user can invoke to analyze a program written in a C-like language: Assume() refers to a label in the source code or to a program part, and enables the user to make an assumption about the state of the program at some label or the function of some program part; Capture() refers to a label or a program part and returns an assertion about the state of the program at the label or the function of the program part; Verify() refers to a label or a program part and tests a unary assertion about the state of the program at the label or a binary assertion about the function of the program part; Establish() refers to a label or a program part and modifies the program code to make Verify() return TRUE at that label or program part, if it did not originally. We discuss the foundations of this tool as well as a preliminary implementation.

Keywords–Assume(), Capture(), Verify(), Establish(), Symbolic execution, while loops, Mathematica (©Wolfran Research), invariant relations.

1. Introduction: Querying a Program at Scale

Despite decades of research in programming language design and implementation, and despite the emergence of many programming languages that have advanced, sophisticated technical attributes, most software being developed, maintained and reused today is written in C-like languages. The six top languages in the July 2023 Tiobe classification (`https://www.tiobe.com/`) of programming languages are derived from or inspired by C. As software maintenance and evolution continue to account for a large, and growing, percentage of software engineering costs and resources, and as software is increasingly developed from existing code, the ability to analyze the function of a software artifact from a static inspection of its source code becomes increasingly critical. The recent talk of using artificial intelligence to generate code makes this capability even more critical because AI code generation is rather opaque, thereby precluding any process-based quality controls.

The question of deriving the function of a program written in a C-like language has eluded researchers for decades, primarily due to the presence of loops, whose function cannot be easily modeled in general. In this paper we see how we can, under some conditions, capture the function of iterative statements, such as while loops, for loops, repeat loops, etc, at arbitrary levels of nesting.

But deriving the function of a program in all its minute detail may be too much information for an analyst to handle; a programmer who abhors poring over pages of source code will probably not relish the prospect of poring over pages of mathematical notation instead. Hence in addition to the ability to compute the function of a program, we are interested to offer the user the ability to query the program at scale. To this effect, we propose four functions, which are invoked in the context of an interactive session:

- *Assume()*.
  - *@L: Assume(C)*, where L is a label and C is a unary predicate on the state of the program, formulates an assumption that the user makes about the state of the program at label L; in particular, this function can be used to formulate the pre-specification of a program or a subprogram.
  - *@P: Assume(C)*, where P is a named program part and C is a binary predicate on the state of the program, formulates an assumption that the user makes about the function of P.
- *Capture()*.
  - *@L: Capture()*, where L is a label, calls on the system to generate two unary conditions: A *Reachability Condition*, which is the condition on the initial state of the program under which execution reaches label L; and a *State Assertion*, which captures everything that is known about the state of the program at label L.
  - *@P: Capture()*, where P is a program part, calls on the system to generate a binary predicate in the state of the program (in (s, s')), which captures everything that is known about the function of P.
- *Verify()*.
  - *@L: Verify(C)*, where L is a label and C is a unary condition on the state of the program, calls on the system to return TRUE of FALSE depending on whether condition C is assured to be true at label L or not. A user may invoke *Verify(C)* to check a program's postcondition for correctness.
  - *@P: Verify(C)*, where P is a program part and C is a binary condition on the state of the program, calls on the

```
public class Main
  {public static int f(int x){x=7*x+7;return x;}
   public static void main(String argv[])
     {int x,y,t,i,j,k; //read x,y,i,j,k,t;
        Label L1; t= i-j; j= i+5;
        if (i>j)
          {x= 0; y= f(x);
           while (i!=j)
             {i=i+k; k=k+1; i=i-k; y=f(y);
               Label L2;} Label L3;}
        else
          {if (j>i)
            {while (j != i)
              {j=j+k; k=k-1; j=j-k; y=f(y);};
             Label L4;}
           else
             {while (t!=i)
               {for(int z=0;z!=y;z=z+1){x=x+1;}
                y= x-y; t= t+1;} Label L5;}}
        k=i+j; j=2*k; Label L6;}}
```

Figure 1. Sample Java Code

system to return TRUE of FALSE depending on whether program P refines the binary relation defined by C.

- *Establish()*.
  - *@L: Establish(C)*, where L is a label and C is a unary condition that does not hold at label L. The call to this function deploys program repair techniques [1] to generate mutants of the path from the first executable statement of the program to label L, and selects a mutant that makes condition C true at label L.
  - *@P: Establish(C)*, where P is a program part and C is a binary predicate on the state of the program that is not refined by the function of P. The call to this function deploys program repair technology to generate mutants of P and select a mutant that makes *@P: Verify(C)* return TRUE.

Due to space limitations, we content ourselves with a brief demo, and refer the interested reader to a video that illustrates the execution of our tool, which we call *ACVE*. **http://web.njit.edu/˜mili/acvedemo.mp4**.

2. System Specification Through a Use Case

The queries we present in this section apply to the code shown in Figure 1. Due to space limitations, we merely show the queries and ACVE's responses, and leave it to the reader to check the validity of these responses.

2.1 @L1: Assume($i >= 0$)

Label $L1$ is located right after the program's inputs are read; hence we interpret this query as the pre-condition of this program. ACVE takes note:

```
(i>=0) assumed at label L1.
```

2.2 @L2: Capture()

We want to know: under what condition label $L2$ is reached and what is known about the state of the program when execution reaches it. The system replies:

```
Reachability Condition:
j>=0 && i>j  || j<0 && i>=0
State Assertion:
1+i==j+t&&x==0&&y==56&&t>0&&(j>=0||j+t>=0)
```

2.3 @L3: Capture()

We want to enquire about the state of the program at label $L3$. The system replies:

```
Reachability Condition:
i>=0 && (j<0 || i>j)
State Assertion:
i==j && x==0 && 7^(2+t)==7+6*y
     && t>0  && (j>=0||j+t>=0)
```

2.4 @L3: Verify()

If we find the output of *Capture()* too detailed and just wants to check a minimal (e.g. safety) condition, then we submit *Verify()* queries. Below are three simple examples.

```
@L3:  Verify(t+j>=0)
>> TRUE
@L3:  Verify(i!=j)
>> FALSE
@L3:  Verify(j>0)
>> FALSE
```

2.5 @L4: Capture()

We submit query *@L4: Capture()* to enquire on the reachability condition of label $L4$, and what is known at that label. The system replies:

```
Reachability Condition:
>> FALSE
State Assertion:
>> FALSE
```

2.6 @L4: Verify()

A query such as: *@L: Verify(C)* means: if execution reaches label L, does condition C hold for the program state at label L? Hence if label L is unreachable, then *@L: Verify(C)* ought to return TRUE for any C. Indeed,

```
@L4:  Verify(1==1)
>> TRUE
@L4:  Verify(1==0)
>> TRUE
```

2.7 @L5: Capture()

To query the program at this label $L5$, after execution of the nested loop, we use *Capture()*. The system replies (where Fib is the Fibonacci function):

```
Reachability Condition
   (i==j) && (j>=0)
State Assertion
   (i==t&&j==t&&(y==(x*Fib[t])/Fib[t+1]&&t>0)
   ||   (t==0&&i==0&&j==0)
```

Acknowledgement

This work is partially supported by NSF grant DGE2043104.

References

[1] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Trans. on Soft. Eng.*, 45(1), January 2019.