Tutorial for P-Gen

Mohamed Nassim Seghir

University of Oxford

This is a brief tutorial that covers some basics of P-Gen.

1 Generating simple preconditions

Let us consider the routine memcpy from the string.h library, an implementation of it is illustrated in Figure 1. This routine copies n bytes from the object pointed by s2 into the object pointed to by s1. We want to verify that the write access to the destination memory region does not go beyond the buffer reserved for the object pointed by s1. As we do not know the upper bound for the memory region reserved for the destination object (pointed by s1), we express it using the variable s1_UB which is just used for the verification purpose. This variable can have any arbitrary value as it is not initialized. The access violation to the buffer pointed by s1 is expressed by the conditional statement at line 14 which is the only way to reach the error location ERROR_1. We call P-Gen via the command

./p-gen –reach –mainproc memcpy –file ../examples/memcpy.c –beyondwp –precond –noinline –tprover z3

The option reach indicates that we are performing a reachability analysis. Options mainproc and file specify the procedure and file name respectively. The most important options here are precond and beyondwp. The first one tells P-Gen to generate the precondition and the second one is for using the inference-rule-based refinement mechanism. Finally, noinline is to avoid renaming local variables of the procedure and tprover is for specifying the theorem prover to use (z3 in our case).

Figure 2 displays the result of P-Gen after it terminates. It shows, in addition to some statistics, the inferred precondition which is $n - 1 < 0 \vee -dst\_UB + s1 + n - 2 < 0$. The first disjunct represents the case where we skip the loop. The second disjunct can be rewritten as $s1 + n - 1 \leq dst\_UB$, it represents the constraint that dst_UB must fulfill when entering to the loop.

2 Generating quantified preconditions

Let us now consider the procedure strlen, illustrated in Figure 3. This procedure returns the length of a 0 terminal string. The statement at line 10 specifies the safe access for each element of the string pointed by s, assuming that s is not equal to NULL. As in the previous example, here also we use s_UB to model the upper bound for the memory occupied by the object which is pointed by s.

```

void * memcpy(void * s1, const void * s2, unsigned int n)
{
    char *dst;
    const char *src;
    // dst_UB is just used for verification purpose
    int dst_UB;

    dst = s1;
    src = s2;
    while (n > 0)
      {
        n = n - 1 ;
        if(dst > s1_UB) goto ERROR_1;
        *dst++ = *src++;

      }
    return s1;

goto end;
ERROR_1:;
end:;
```

Fig. 1. Program that copies a memory reggion to another memory region.

In this case, the content of the buffer (string) is used to mark its bound ('\0'). Thus, the content is relevant to the property that we want to verify. This time we call P-Gen using the command

./p-gen –reach –mainproc strlen –file ../examples/strlen.c –beyondwp –precond –noinline –tprover z3 –stringabs –dontabsarray –expheap

We can see three new options: stringabs, dontabsarray and expheap. The option stringabs indicates to P-Gen to abstract string and character constants using freshly generated identifiers. This is required as we do not have a dedicated decision procedure. The option expheap is used to explicitly represent the heap as an array. Finally, dontabsarray is used to avoid abstracting arrays in conditional statements. As the heap is modeled as an array and it is relevant to the property we want to verify, it makes sense not to abstract arrays. After calling P-Gen, we obtain the result displayed in Figure 4. The identifier ABS_STR_1 is used to model the character '\0'. The heap is explicitly modeled as an array of name ACS_HEAP, hence ACS_HEAP[s] models *s. The obtained precondition says that either s points to the character '\0' or the character '\0' must be pointed by an element from the interval $[s + 1, s\_UB]$.

```
INFORMATION ABOUT THE COMPUTATION OF UNSAFE STATES
NUMBER OF ITERATIONS: 4
Average predicate number per location: 2
Remain: 3
Nbr Loc: 5


 INFORMATION ABOUT THE COMPUTATION OF SAFE STATES
 NUMBER OF ITERATIONS: 5
Average predicate number per location: 2
Remain: 2
Nbr Loc: 6



***********          COMPUTED PRECONDITION         *********
(n- 1 < 0 || -dst_UB+ s1+ n- 2 < 0 )
***********************************************************
```

Fig. 2. Result returned by P-Gen for procedure memcpy.

```

unsigned int strlen(const char *s)
 {
     int s_UB;
     p = s;

     while (*p != '\0')
       {
         p++;
         if(p  > s_UB) goto ERROR_1;
       }
    return (size_t)(p - s);
    goto end;
 ERROR_1:;
 end:;

}
```

Fig. 3. Program that computes the length of a 0 terminal string.

3 Displaying internal information

One can display some additional internal information stored by P-Gen. The option printcmd allows to display the internal representation of programs as transition constraints, for the program strcpy we obtain the result in Figure 5.

```
INFORMATION ABOUT THE COMPUTATION OF UNSAFE STATES
NUMBER OF ITERATIONS: 4
Average predicate number per location: 4
Remain: 0
Nbr Loc: 3


INFORMATION ABOUT THE COMPUTATION OF SAFE STATES
NUMBER OF ITERATIONS: 4
Average predicate number per location: 3
Remain: 2
Nbr Loc: 3


***********          COMPUTED PRECONDITION         *********
 (-ABS_STR_1+ ACS_HEAP[ s] == 0 ||
 !FORALL( -ABS_STR_1+ ACS_HEAP[ UNI_1] != 0 , UNI_1>= s+ 1 , UNI_1<= s_UB ) )
************************************************************
```

Fig. 4. Result returned by P-Gen for procedure strlen.

We can also display the abstraction map (location to predicate set) using option

```
pc== 1 && pc_1== 3 && dst== s1&& src== s2 [][  ]
pc== 5 && dst<= dst_UB&& pc_1== 3 && dst== dst+ 1 && src== src+ 1 && *dst== *src+ 1  [][  ]
pc== 3 && n> 0 && pc_1== 5 && n== n- 1  [][  ]
pc== 3 && n<= 0 && pc_1== 6  [][  ]
pc== 6 && pc_1== 7  [][  ]
pc== 11 && pc_1== 7  [][  ]
pc== 5 && dst> dst_UB&& pc_1== 8  [][  ]
```

Fig. 5. List of transition constraints corresponding to program memcpy.

abstmap to get the result shown in Figure 6.

```
Abstraction map:

location: 1 --> [0,n- 1 >= 0 ][5,-dst_UB+ s1- 1 >= 0 ][9,-dst_UB+ s1+ n- 2 >= 0 ]
location: 2 --> [0,n- 1 >= 0 ][5,-dst_UB+ s1- 1 >= 0 ]
location: 3 --> [0,n- 1 >= 0 ][2,dst- dst_UB- 1 >= 0 ][10,dst- dst_UB+ n- 2 >= 0 ]
location: 4 --> [0,n- 1 >= 0 ][9,-dst_UB+ s1+ n- 2 >= 0 ]
location: 5 --> [2,dst- dst_UB- 1 >= 0 ][11,dst- dst_UB+ n- 1 >= 0 ]
```

Fig. 6. Abstraction map corresponding to program memcpy.