# Chapter 7

# Symbolic Execution and Proof of Properties

Symbolic execution builds predicates that characterize the conditions under which execution paths can be taken and the effect of the execution on program state. Extracting predicates through symbolic execution is the essential bridge from the complexity of program behavior to the simpler and more orderly world of logic. It finds important applications in program analysis, in generating test data, and in formal verification[1] (proofs) of program correctness.

Conditions under which a particular control flow path is taken can be determined through symbolic execution. This is useful for identifying infeasible program paths (those that can never be taken) and paths that could be taken when they should not. It is fundamental to generating test data to execute particular parts and paths in a program.

Deriving a logical representation of the effect of execution is essential in methods that compare a program's possible behavior to a formal specification. We have noted in earlier chapters that proving the correctness of a program is seldom an achievable or useful goal. Nonetheless the basic methods of formal verification, including symbolic execution, underpin practical techniques in software analysis and testing. Symbolic execution and the techniques of formal verification find use in several domains:

- Rigorous proofs of properties of (small) critical sub-systems, such as a safety kernel of a medical device;
- Formal verification of critical properties (e.g., security properties) that are particularly resistant to dynamic testing;
- Formal verification of algorithm descriptions and logical designs that are much less complex than their implementations in program code.

[1]Throughout this book we use the term *verification* in the broad sense of checking whether a program or system is consistent with some form of specification. The broad sense of verification includes, for example, inspection techniques and program testing against informally stated specifications. The term *formal verification* is used in the scientific literature in a much narrower sense to denote techniques that construct a mathematical proof of consistency between some formal representation of a program or design and a formal specification.

More fundamentally, the techniques of formal reasoning are a conceptual foundation for a variety of analysis techniques, ranging from informal reasoning about program behavior and correctness to automated checks for certain classes of errors.

## 7.1 Symbolic State and Interpretation

Tracing execution is familiar to any programmer who has attempted to understand the behavior of source code by simulating execution. For example, one might trace a single statement in the binary search routine of Figure 7.1 as shown on the left side of Figure 7.2. One can just as easily use symbolic values like $L$ and $H$ in place of concrete values, as shown on the right side of Figure 7.2. Tracing execution with symbolic values and expressions is the basis of symbolic execution.

When tracing execution with concrete values, it is clear enough what to do with a branch statement, e.g., an if of while test: The test predicate is evaluated with the current values, and the appropriate branch is taken. If the values bound to variables are symbolic expressions, however, both the *True* and *False* outcomes of the decision may be possible. Execution can be traced through the branch in either direction, and execution of the test is interpreted as adding a constraint to record the outcome. For example, consider

```
while (high >= low) {
```

Suppose the symbolic state after one loop iteration is

$$\begin{array}{ll} & \text{low} = 0 \\ \wedge & \text{high} = \frac{H-1}{2} - 1 \\ \wedge & \text{mid} = \frac{H-1}{2} \end{array}$$

If we trace execution of the test assuming a *True* outcome (leading to a second iteration of the loop), the loop condition becomes a constraint in the the symbolic state immediately after the while test:

$$\begin{array}{ll} & \text{low} = 0 \\ \wedge & \text{high} = \frac{H-1}{2} - 1 \\ \wedge & \text{mid} = \frac{H-1}{2} \\ \wedge & \frac{H-1}{2} - 1 \geq 0 \end{array}$$

Later, when we consider the branch assuming a *False* outcome of the test, the new constraint is negated and becomes $\neg(\frac{H-1}{2} - 1 \geq 0)$ or, equivalently, $\frac{H-1}{2} - 1 < 0$.

Execution can proceed in this way down any path in the program. One can think of "satisfying" the predicate by finding concrete values for the symbolic variables that make it evaluate to *True*; this corresponds to finding data values that would force execution of that program execution path. If no such satisfying values are possible, then that execution path cannot be executed with any data values; we say it is an infeasible path.

```
/** Binary search for key in sorted array dictKeys, returning
*  corresponding value from dictValues or null if key does
*  not appear in dictKeys.  Standard binary search algorithm
*  as described in any elementary text on data structures and algorithms.
**/

char * binarySearch( char *key, char *dictKeys[ ], char *dictValues[ ],
                                    int dictSize) {

   int low = 0;
   int high = dictSize - 1;
   int mid;
   int comparison;

   while (high >= low) {
      mid = (high + low) / 2;
      comparison = strcmp( dictKeys[mid], key );
      if (comparison < 0) {
         /* dictKeys[mid] too small; look higher */
           low = mid + 1;
      } else if ( comparison > 0 ) {
         /* dictKeys[mid] too large; look lower */
           high = mid - 1;
      } else {
         /* found */
           return dictValues[mid];
      }
   }
   return 0;     /* null means not found */
}
```

Figure 7.1: Binary search procedure

*before*

| | |
|---|---|
| low | 12 |
| high | 15 |
| mid | – |

```
mid = (high + low) / 2;
```

*after*

| | |
|---|---|
| low | 12 |
| high | 15 |
| mid | 13 |

*before*

| | |
|---|---|
| low | $L$ |
| high | $H$ |
| mid | – |

```
mid = (high + low) / 2;
```

*after*

| | |
|---|---|
| low | $L$ |
| high | $H$ |
| mid | $\frac{L+H}{2}$ |

Figure 7.2: Hand-tracing an execution step with concrete values (left) and symbolic values (right).

## 7.2 Summary Information

If there were only a finite number of execution paths in a program, then in principle a symbolic executor could trace each of them and obtain a precise representation of a predicate that characterizes each one. From even a few execution steps in the small example above, one can see that the representation of program state will quickly become unwieldy. Moreover, there are a potentially infinite number of program execution paths to consider. An automated symbolic executor can cope with much more complex symbolic expressions than a human, but even an automated tool will not get far with brute force evaluation of every program path.

Since the representation of program state is a logical predicate, there is an alternative to keeping a complete representation of the state at every point: a *weaker* predicate can always be substituted for the complete representation. That is, if the representation of the program state at some point in execution is $P$, and if $W \Rightarrow P$, then substituting $W$ for $P$ will result in a predicate that still correctly describes the execution state, but with less precision.

Consider the computation of mid in line 17 of the binary search example from Figure 7.1. If we are reasoning about the performance of binary search, the fact that the value of mid lies half-way between the values of low and high is important, but if we are reasoning about functional correctness it matters only that mid lies somewhere between them. Thus, if we had $\text{low} = L \wedge \text{high} = H \wedge \text{mid} = M$, and if we could show $L \leq H$, we could replace $M = (L+H)/2$ by the weaker condition $L \leq M \leq H$.

Note that the weaker predicate $L \leq \text{mid} \leq H$ is chosen based on what must be true for the program to execute correctly. This is not information that can be derived automatically from source code; it depends as well on our understanding of the code and our rationale for believing it to be correct. A predicate stating what *should* be true at a given point can be expressed in the form of an assertion. When we assert that predicate $W$ is true at a point in a program, we mark our intention both to verify it at that point (by showing that $W$ is implied by the predicates that describe the program state at that point) and to replace part of the program state description $P$ by $W$ at that point.

One of the prices of weakening the predicate in this way will be that satisfying the predicate is no longer sufficient to find data that forces the program execution along that path. If the complete predicate $P$ is replaced by a weaker predicate $W$, then test data that satisfies $W$ is necessary to execute the path, but it may not be sufficient. Showing that $W$ cannot be satisfied is still tantamount to showing that the execution path is infeasible.

## 7.3 Loops and Assertions

The number of execution paths through a program with one or more loops is potentially infinite, or at least unimaginably huge. This may not matter for symbolic execution along a single, relatively simple execution path. It becomes a major obstacle if symbolic execution is used to reason about a path involving several iterations of a loop, or to reason about all possible program executions.

To reason about program behavior in a loop, we can place within the loop an assertion that states a predicate expected to be true each time execution reaches that point. Such an assertion is called an *invariant*. Δ loop invariant Each time program execution reaches the invariant assertion, we can weaken the description of program state. If the program state is represented by $P$, and the assertion is $W$, we must first ascertain $W \Rightarrow P$ (the assertion is satisfied along that path), and then we can substitute $W$ for $P$.

Suppose every loop contained such an assertion, and suppose in addition there was an assertion at the beginning of the program (perhaps just the trivial predicate *True*) and one at the end. In that case, every possible execution path would consist of a sequence of segments from one assertion to the next. The assertion at the beginning of a segment is the *precondition* for that segment, Δ precondition and the assertion at the end of the segment is the *postcondition*. Δ postcondition If we were able to execute each such segment independently, starting with only the precondition and then checking that the assertion at the end of the segment is satisfied, we would have shown that every assertion is satisfied on every possible program execution — i.e., we would have verified correct execution on an infinite number of program paths, by verifying the finite number of segments from which the paths are constructed.

We illustrate the technique by using assertions to check the logic of the binary search algorithm implemented by the program in Figure 7.1. The first precondition and the final postcondition serve as a specification of correct behavior as a kind of contract: If the client ensures the precondition, the program will ensure the postcondition.

The binary search procedure depends on the array dictKeys being sorted. Thus we might have a precondition assertion like the following:

$$\forall i, j, 0 \leq i < j < \text{size} : dictKeys[i] \leq dictKeys[j]$$

Here we interpret $s \leq t$ for strings as indicating lexical order consistent with the C library strcmp, i.e., we assume that $s \leq t$ whenever strcmp(s,t) $\leq 0$. For convenience we will abbreviate the predicate above as *sorted*.

We can associate the following assertion with the while statement at line 16:

$$\forall i, 0 \leq i < \text{size} : \text{dictkeys}[i] = \text{key} \Rightarrow \text{low} \leq i \leq \text{high}$$

In other words, we assert that the key can appear only between low and high, if it appears anywhere in the array. We will abbreviate this condition as *inrange*.

*Inrange* must be true when we first reach the loop, because at that point the range $\text{low} \ldots \text{high}$ is the same as $0 \ldots \text{size} - 1$. For each path through the body of the loop, the symbolic executor would begin with the invariant assertion above, and determine that it is true again after following that path. We say the invariant is *preserved*.

While the *inrange* predicate should be true on each iteration, it is not the complete loop invariant. The *sorted* predicate remains true and will be used in reasoning. In principle it is also part of the invariant, although in informal reasoning we may not bother to write it down repeatedly. The full invariant is therefore $\textit{sorted} \wedge \textit{inrange}$.

Let us consider the path from line 16 through line 21 and back to the loop test. We begin by assuming the loop invariant assertion holds at the beginning of the segment. Where expressions in the invariant refer to program variables whose values may change, they are replaced by symbols representing the initial values of those variables. The variable bindings will be

$$\begin{array}{ll} & \text{low} = L \\ \wedge & \text{high} = H \end{array}$$

We need not introduce symbols to represent the values of dictKeys, dictVals, key, or size. Since those variables are not changed in the procedure, we can use the variable names directly. The condition, instantiated with symbolic values, will be

$$\begin{array}{ll} & \forall i, j, 0 \leq i < j < \text{size} : \textit{dictKeys}[i] \leq \textit{dictKeys}[j] \\ \wedge & \forall k, 0 \leq k < \text{size} : \text{dictkeys}[k] = \text{key} \Rightarrow L \leq k \leq H \end{array}$$

Passing through the while test into the body of the loop adds the condition $H \geq L$ to the condition above. Execution of line 17 adds a binding of $\lfloor (H+L)/2 \rfloor$ to variable mid, where $\lfloor x \rfloor$ is the integer obtained by rounding $x$ toward zero. As we have discussed, this can be simplified with an assertion so that the bindings and condition become

$$\begin{array}{lll} & \text{low} = L & \textit{(bindings)} \\ \wedge & \text{high} = H & \\ \wedge & \text{mid} = M & \\ \wedge & \forall i, j, 0 \leq i < j < \text{size} : \text{dictKeys}[i] \leq \text{dictKeys}[j] & \textit{(sorted)} \\ \wedge & \forall k, 0 \leq k < \text{size} : \text{dictkeys}[k] = \text{key} \Rightarrow L \leq k \leq H & \textit{(inrange)} \\ \wedge & H \geq M \geq L & \end{array}$$

Tracing the execution path into the first branch of the if statement to line 21, we add the constraint that strcmp(dictKeys[mid], key) returns a negative value, which we interpret as meaning the probed entry is lexically less than the string value of the key. Thus we arrive at the symbolic constraint

$$\begin{array}{ll} & \text{low} = L \\ \wedge & \text{high} = H \\ \wedge & \text{mid} = M \\ \wedge & \forall i,j, 0 \leq i < j < \text{size} : \text{dictKeys}[i] \leq \text{dictKeys}[j] \\ \wedge & \forall k, 0 \leq k < \text{size} : \text{dictkeys}[k] = \text{key} \Rightarrow L \leq k \leq H \\ \wedge & H \geq M \geq L \\ \wedge & \text{dictKeys}[M] < \text{key} \end{array}$$

The assignment in line 21 then modifies a variable binding without otherwise disturbing the conditions, giving us

$$\begin{array}{ll} & \text{low} = M+1 \\ \wedge & \text{high} = H \\ \wedge & \text{mid} = M \\ \wedge & \forall i,j, 0 \leq i < j < \text{size} : \text{dictKeys}[i] \leq \text{dictKeys}[j] \\ \wedge & \forall k, 0 \leq k < \text{size} : \text{dictkeys}[k] = \text{key} \Rightarrow L \leq k \leq H \\ \wedge & H \geq M \geq L \\ \wedge & \text{dictKeys}[M] < \text{key} \end{array}$$

Finally, we trace execution back to the while test at line 16. Now our obligation is to show that the invariant still holds when instantiated with the changed set of variable bindings. The *sorted* condition has not changed, and showing that it is still true is trivial. The interesting part is the *inrange* predicate, which is instantiated with a new value for low and thus becomes

$$\forall k, 0 \leq k < \text{size} : \text{dictkeys}[k] = \text{key} \Rightarrow M+1 \leq k \leq H$$

Now the verification step is to show that this predicate is a logical consequence of the predicate describing the program state. This step requires purely logical and mathematical reasoning, and might be carried out either by a human or by a theorem-proving tool. It no longer depends in any way upon the program. The task performed by the symbolic executor is essentially to transform a question about a program (is the invariant preserved on a particular path?) into a question of logic alone.

The path through the loop on which the probed key is too large, rather than too small, proceeds similarly. The path on which the probed key matches the sought key returns from the procedure, and our obligation there (trivial in this case) is to verify that the contract of the procedure has been met.

The other exit from the procedure occurs when the loop terminates without locating a matching key. The contract of the procedure is that it should return the null pointer (represented in the C language by 0) only if the key appears nowhere in dictKeys[0..size-1]. Since the null pointer is returned whenever the loop terminates, the postcondition of the loop is that key is not present in dictKeys.

The loop invariant is used to show that the postcondition holds when the loop terminates. What symbolic execution can verify immediately after a loop is that the invariant is true but the loop test is false. Thus we have

$$\begin{array}{lll} & \text{low} = L & \textit{(bindings)} \\ \wedge & \text{high} = H & \\ \wedge & \forall i,j, 0 \leq i < j < \text{size} : \text{dictKeys}[i] \leq \text{dictKeys}[j] & \textit{(sorted)} \\ \wedge & \forall k, 0 \leq k < \text{size} : \text{dictkeys}[k] = \text{key} \Rightarrow L \leq k \leq H & \textit{(inrange)} \\ \wedge & L > H & \end{array}$$

Knowing that presence of the key in the array implies $L \leq H$, and that in fact $L > H$, we can conclude that the key is not present. Thus the postcondition is established, and the procedure fulfills its contract by returning the null pointer in this case.

Finding and verifying a complete set of assertions, including an invariant assertion for each loop, is difficult in practice. Even the small example above is rather tedious to verify by hand. More realistic examples can be quite demanding even with the aid of symbolic execution tools. If it were easy or could be fully automated, we might routinely use this method to prove the correctness of programs. Writing down a full set of assertions formally, and rigorously verifying them, is usually reserved for small and extremely critical modules, but the basic approach we describe here can also be applied in a much less formal manner, and is quite useful in finding holes in an informal correctness argument.

## 7.4 Compositional Reasoning

The binary search procedure is very simple. There is only one loop, containing a single if statement. It was not difficult to reason about individual paths through the control flow. If the procedure contained nested loops or more conditional branches, we could in principle still proceed in that manner as long as each cycle in the control flow graph were broken by at least one assertion. It would, however, be very difficult to think about programs in this manner, and to choose appropriate assertions. It is better if our approach follows the hierarchical structure of the program, both at a small scale (e.g., control flow within a single procedure) and at larger scales (across multiple procedures, classes, subsystems, etc.).

The steps for verifying the binary search procedure above already hint at a hierarchical approach. The loop invariant was not placed just anywhere in the loop. We associated it with the beginning of the loop so that we could follow a standard style of reasoning that allows us to compose facts about individual pieces of a program to derive facts about larger pieces. In this hierarchical or compositional style, the effect of any program block is described by a *Hoare triple*:

Hoare triple

$$(\!|\, pre \,|\!) \ \text{block} \ (\!|\, post \,|\!)$$

The meaning of this triple is that if the program is in a state satisfying the precondition *pre* at entry to the block, then after execution of the block it will be in a state satisfying the postcondition *post*.

There are standard templates, or schemata, for reasoning with triples. In the previous section we were following this schema for reasoning about while loops:

$$\frac{(\!| I \wedge C |\!)\ \ S\ \ (\!| I |\!)}{(\!| I |\!)\ \ \text{while(C) \{ S \}}\ \ (\!| I \wedge \neg C |\!)}$$

The formula above the line is the premise of an inference, and the formula below the line is the conclusion. An inference rule states that if we can verify the premise, then we can infer the conclusion. The premise of this inference rule says that the loop body preserves invariant $I$: If the invariant $I$ is true before the loop, and if the condition $C$ governing the loop is also true, then the invariant is established again after executing the loop body $S$. The conclusion says that the loop as a whole takes the program from a state in which the invariant is true to a state satisfying a postcondition composed of the invariant and the negation of the loop condition.

The important characteristic of these rules is that they allow us to compose proofs about small parts of the program into proofs about larger parts. The inference rule for while allows us to take a triple about the body of a loop and infer a triple about the whole loop. There are similar rules for building up triples describing other kinds of program blocks. For example:

$$\frac{(\!| P \wedge C |\!)\ \ \mathit{thenpart}\ \ (\!| Q |\!) \qquad (\!| P \wedge \neg C |\!)\ \ \mathit{elsepart}\ \ (\!| Q |\!)}{(\!| P |\!)\ \ \text{if (C) \{}\mathit{thenpart}\ \text{\} else \{}\ \mathit{elsepart}\ \text{\}}\ \ (\!| Q |\!)}$$

This style of reasoning essentially lets us summarize the effect of a block of program code by a precondition and a postcondition. Most importantly, we can summarize the effect of a whole procedure in the same way. The *contract* of the procedure is a precondition (what the calling client is required to provide) and a postcondition (what the called procedure promises to establish or return). Once we have characterized the contract of a procedure in this way, we can use that contract wherever the procedure is called. For example, we might summarize the effect of the binary search procedure this way:

$$(\!| \forall i, j, 0 \le i < j < \text{size} : \text{keys}[i] \le \text{keys}[j] |\!)$$

s = binarySearch(k, keys, vals, size)

$$(\!| \begin{array}{ll} & (\text{s} = v \wedge \exists i, 0 \le i < \text{size} : \text{keys}[i] = \text{k} \wedge \text{vals}[i] = v) \\ \vee & (\text{s} = 0 \wedge \nexists i, 0 \le i < \text{size} : \text{keys}[i] = \text{k}) \end{array} |\!)$$

## 7.5 Reasoning about Data Structures and Classes

The contract of the binary search procedure can be specified in a relatively simple, self-contained manner. Imagine, though, that it is part of a module that maintains a dictionary structure, e.g., the relation between postal codes and the nearest airport with air-freight capability. In that case, the responsibility for keeping the table in sorted order would belong to the module itself, and not to its clients. If implemented in a modern object-oriented language, the data structure would not even be visible to the client, but would rather be encapsulated within a class.