Symbolic Execution

- Process of building predicates that describe which execution paths will be taken and their effect on program state.
  - Determines the conditions under which a path can be taken.
  - Identifies infeasible paths and paths that can be taken when they shouldn’t.
  - Can be used to generate tests targeted at particular paths in the system.
Symbolic Execution

- Bridge between complex program behavior and analyzable logical structures.
  - Enables complex analyses of programs through abstraction to a model of execution.
  - Allows proof of properties over small critical subsystems.
  - Allows formal verification of critical properties resistant to testing.
  - Allows formal verification of logical designs before code is written.
What is Symbolic Execution?

Program Execution
- Execute the program with actual values.
- Statements compute new values for variables.
- Program state can be characterized by the values of variables.

Symbolic Execution
- Execute the program with symbolic values
- Statements compute new symbolic expressions
- Program state can be characterized by predicates made of symbolic expressions
Assigning Meaning to Programs

```c
void example(int n)
{
    int i = 1;
    int S = 0;
    if (i > n)
        return;
    else
    {
        S = S + i;
        i = i + 1;
    }
}
```

- We increase the value of S by i.
- i increases by 1.
- Therefore, S is a summation over 1 to i
Binary Search

```c
char *binarySearch( char *key, char *dictKeys[], char *dictValues[], int dictSize ) {
    int low = 0;
    int high = dictSize - 1;
    int mid, comparison;
    while (high >= low) {
        mid = (high + low) / 2;
        comparison = strcmp( dictKeys[mid], key );
        if (comparison < 0) {
            low = mid + 1;
        } else if ( comparison > 0 ) {
            high = mid - 1;
        } else {
            return dictValues[mid];
        }
    }
    return 0;
}
```
Effect of Executing a Statement

\[
\text{mid} = \frac{(\text{low} + \text{high})}{2};
\]

Concrete Values
- **Before:**
  - low = 8 ^ high = 13
- **After:**
  - low = 8 ^ high = 13 ^ mid = 10

Symbolic Values
- **Before:**
  - low = L ^ high = H
- **After:**
  - low = L ^ high = H ^ mid = (L + H) / 2
Dealing with Branches

- Taking a branch adds a constraint to the program state.
- Add that constraint to the predicate describing the state.

\[
\text{low} = L \land \text{high} = H \\
\land \neg(H \geq L) \\
\text{(or H < L)}
\]

\[
\text{high} \geq \text{low}
\]
Symbolic Execution

- “Satisfying the predicate” can mean finding concrete values that make it evaluate to true.
  - This is a test case forcing the program to take a path. If no values can be found, then this is an infeasible path.
- If there are a finite number of paths in a program, a symbolic executor can trace each and obtain predicates characterizing each one.
Symbolic representation of state can easily grow too complex to use.
  - And potentially an infinite number of paths.

Can simplify the property we are checking:
  - P characterizes a state.
  - P => W
    - W is a simpler predicate than P.
  - We can use W instead of P.
    - W is a summary of P.
Example: Summary Information

\[ \text{mid} = (\text{low} + \text{high}) / 2; \]

**Symbolic Values**

- **Before:**
  - \( \text{low} = L \land \text{high} = H \)

- **After:**
  - \( \text{low} = L \land \text{high} = H \land \text{mid} = (L + H) / 2 \)

\[ \text{mid} = M \land H \geq M \geq L \]
Assertions

● Weaker predicate based on what must be true for the program to execute correctly.
  ○ Cannot be derived automatically.
● Also known as an **assertion**.
  ○ A predicate stating what *should* be true at a particular point in program execution.
● Making an assertion marks our intention to verify that the predicate is true.
  ○ and that it is acceptable to replace part of the state with that property.
Effect of Weakening

- Required at times to make symbolic execution possible for complex programs.
- That predicate is no longer sufficient to find input that forces execution along that path.
  - Satisfying that predicate is *necessary but not sufficient* to exercise the path.
  - Showing that the predicate cannot be satisfied still shows that the path is infeasible.
Working with Loops

- Number of paths is infinite in the presence of loops.
- To reason with loops in symbolic execution:
  - Use a summary (assertion) to describes the program state when control reaches the loop.
    - Called a loop invariant.
  - Does not change based on the number of iterations.
  - When execution reaches the invariant, we check that the loop invariant is true at that point.
Verifying Correctness

- Choose a program segment.
  - At the beginning of that segment, place an assertion that must be true (a pre-condition).
  - At the end, place another assertion that must be true (a post-condition).
- Every program path is a sequence of segments from one assertion to the next.
- Verification = ensuring that any possible sequence of segments is logically valid with pre/post-conditions.
char *binarySearch( char *key, char *dictKeys[], char *dictValues[], int dictSize ) {
    int low = 0;
    int high = dictSize - 1;
    int mid, comparison;
    while (high >= low) {
        mid = (high + low) / 2;
        comparison = strcmp( dictKeys[mid], key );
        if (comparison < 0) {
            low = mid + 1;
        } else if (comparison > 0) {
            high = mid - 1;
        } else {
            return dictValues[mid];
        }
    }
    return 0;
}

**pre-condition:** \( \forall \ i, j, 0 \leq i < j < \text{size}: \text{dictKeys}[i] \leq \text{dictKeys}[j] \)

- If the client obeys the pre-condition, the program will obey the post-condition.

**loop invariant:** \( \forall \ i, 0 < i < \text{size}: \text{dictKeys}[i] = \text{key} \Rightarrow \text{low} \leq i < \text{high} \)

- True when we reach the loop.
- True at beginning of each loop cycle.
- True after the end of the loop.
- Symbolic execution begins with the invariant and determines that it is true again following the path.
- The pre-condition must remain true as well.
  - The full loop invariant includes the pre-condition.
Example - Binary Search

while (high >= low) {
    mid = (high + low) / 2;
    comparison = strcmp( dictKeys[mid], key );
    if (comparison < 0) {
        low = mid + 1;
    } else if (comparison > 0) {
        high = mid - 1;
    } else {
        return dictValues[mid];
    }
}

pre-condition (PC): \( \forall \ i, \ j, \ 0 \leq i < j \leq \text{size}: \text{dictKeys}[i] \leq \text{dictKeys}[j] \)

loop invariant (LI): \( \forall \ k, \ 0 < k < \text{size}: \text{dictKeys}[k] = \text{key} \Rightarrow L \leq k < H \)

bindings: \( \text{low} = L \wedge \text{high} = H \)
while (high >= low) {
    mid = (high + low) / 2;
    comparison = strcmp( dictKeys[mid], key );
    if (comparison < 0) {
        low = mid + 1;
    } else if (comparison > 0) {
        high = mid - 1;
    } else {
        return dictValues[mid];
    }
}

pre-condition (PC): ∀ i, j, 0 <= i < j < size: dictKeys[i] <= dictKeys[j]
loop invariant (LI): ∀ k, 0 < k < size: dictKeys[k] = key => L <= k < H
bindings: low = L ^ high = H
Example - Binary Search

while (high >= low) {
    mid = (high + low) / 2;
    comparison = strcmp( dictKeys[mid], key );
    if (comparison < 0) {
        low = mid + 1;
    } else if (comparison > 0) {
        high = mid - 1;
    } else {
        return dictValues[mid];
    }
}

Verify the contract of the procedure:
Returns corresponding value from dictValues for the key in dictKeys, or null if key does not appear in dictKeys.

s=value ^ ∃ i, 0 <= i < size: dictKeys[i] = k ^ dictValues[i] = value

pre-condition (PC): ∀ i, j, 0 <= i < j < size: dictKeys[i] <= dictKeys[j]
loop invariant (LI): ∀ k, 0 < k < size: dictKeys[k] = key => L <= k < H
bindings: low = L ^ high = H
Example - Binary Search

```c
char *binarySearch( char *key, char dictKeys[], char dictValues[], int dictSize ) {
    int low = 0;
    int high = dictSize - 1;
    int mid, comparison;
    while (high >= low) {
        mid = (high + low) / 2;
        comparison = strcmp( dictKeys[mid], key );
        if (comparison < 0) {
            low = mid + 1;
        } else if ( comparison > 0 ) {
            high = mid - 1;
        } else {
            return dictValues[mid];
        }
    }
    return 0;
}
```

**pre-condition (PC):** \( \forall \ i, j, 0 \leq i < j < \text{size}: \text{dictKeys}[i] \leq \text{dictKeys}[j] \)

**loop invariant (LI):** \( \forall \ k, 0 < k < \text{size}: \text{dictKeys}[k] = \text{key} \Rightarrow L \leq k < H \)

**bindings:** low = L \^\^ high = H

**bindings \^ PC \^ LI \^ L>H**

- Presence of the key implies \( L < H \)
- But, \( L > H \)
- Therefore, the key is not present.
- The post-condition is met.

**post-condition:** \( s=0 \^\^ \exists \ a, 0 \leq a < \text{size} : \text{dictKeys}[a] = \text{key} \)

**Verify the contract of the procedure:**
Returns corresponding value from dictValues for the key in dictKeys, or *null if key does not appear in dictKeys.*
The loop body of the binary search can be modified to:

```java
if (comparison < 0){
    low = mid + 1;
}
if (comparison > 0){
    high = mid -1;
}
if (comparison == 0){
    return dictValues[mid];
}
```

Demonstrate using symbolic execution that the path that traverses the false branch of all three statements is infeasible.
if (comparison < 0)
    low = mid + 1;
}
if (comparison > 0)
    high = mid -1;
}
if (comparison == 0)
    return dictValues[mid];
Programs can be structured and verified in a hierarchy of segments.

Loop invariant is placed at beginning of the loop so we can compose facts about pieces of a program.

Effect of a block is described as a *Hoare Triple*:

- $(|pre|) \text{ block } (|post|)$
- If $pre$ is satisfied at entry, then after executing $block$, $post$ will be satisfied.
Inference Rules

- Standard templates for reasoning with triples
- While Loops:

\[
(\|I \land C\|) S (\|I\|) \\
(\|I \land \neg C\|)
\]

Formula on top line is the **premise**.
Formula on the bottom line is the **conclusion**.
If we can verify the premise, we can infer the conclusion.
Inference Rules - While

● While Loops:

\[
\frac{|I \land C| \cdot S \cdot |I|}{|I| \text{ while(C) } \{ S \} \cdot |I \land \neg C|}
\]

● Premise:
  ○ If invariant (I) and loop condition (C) are true before the loop, then after executing the loop body (S), I will still be true.

● Conclusion:
  ○ The loop takes the program from a state where I is true to a state where I is true and C is not.
Inference Rules - If-Statement

\[
(|P \land C|) \text{ thenpart } (|Q|) \ (|P \land \neg C|) \text{ elsepart } (|Q|)
\]

\[
(|P|) \ \text{if}(C) \ \{ \ \text{thenpart} \ \} \ \text{else} \ \{ \ \text{elsepart} \ \} \ (|Q|)
\]

- **Premise:**
  - If pre-condition (P) and if condition (C) are true, then after executing *thenpart* a postcondition (Q) will be true. If P is true and C is false, then after executing *elsepart*, Q is true.

- **Conclusion:**
  - The if-statement takes the program from a state where P is true to a state where Q is true.
Compositional Reasoning

• Can compose proofs about small parts of the program into proofs about larger parts.
  ○ Inference rule for \textit{while} lets us take a triple about the loop body and infer a triple about the whole loop.

• Summarize the effect of a block of code by a pre-condition and post-condition.
  ○ Can summarize the effect of the whole procedure in the same way.
  ○ Establish a \textit{contract} for that block of code.
Compositional Reasoning

● The contract of a procedure is:
  ○ Pre-condition: What the client is required to provide.
  ○ Post-condition: What the procedure promises to establish or return.

● Can use that contract whenever the procedure is called to verify input and results

● Binary Search:
  ○ $(\{ \forall \ i, j, 0 <= i < j < size: \text{dictKeys}[i] <= \text{dictKeys}[j] \})$
  ○ $s = \text{binarySearch}(k, \text{dictKeys}, \text{dictValues}, \text{size})$
  ○ $(\{ (s=value \land \exists \ i, 0 <= i < size: \text{dictKeys}[i] = k \land \text{dictValues}[i] = value) \lor s=0 \land \nexists \ a, 0 <= a < size : \text{dictKeys}[a] = key \})$
Activity 2 - Contract

- The following method calculates the sum of an array of floats.
- Write the pre- and post-conditions for this method.

```java
float sum(int array[], int len) {
    float sum = 0.0;
    int i = 0;
    while (i < length) {
        sum = sum + array[i];
        i = i + 1;
    }
    return sum;
}
```
float sum(int array[], int len) {
    float sum = 0.0;
    int i = 0;
    while (i < length) {
        sum = sum + array[i];
        i = i + 1;
    }
    return sum;
}
Classes and Data Structures

- Classes often maintain data structures.
  - If a method is called on that structure, the responsibility for that structure’s correctness belongs to the class, not the caller.

- Modular verification must obey modular design of the program.
  - Contract cannot reveal private details.
Abstract Model of Data

● Data structure module provides a collection of methods with related specifications.
  ○ Specifications are contracts with clients.
  ○ Specify pre and post-conditions of an abstract model of the encapsulated data.

■ Dictionary:
  ● Contracts in terms of <key, value> pairs.
  ● Actual implementation could be a hashmap, sorted array, tree, etc.
  ● Details of implementation hidden.
  ● Reason over correctness of the abstraction.
Structural Invariants

- Class must preserve properties over the (abstract) data structure it maintains.
  - If structure is sorted arrays, then the class must maintain the sorted order.
  - If structure is balanced search tree, then the class must keep the tree balanced.

- Called *structural invariants*.
  - Similar to loop invariant.
  - Must hold before method invocation and after return.
Abstraction Function

- Behavior must reflect the abstract model.
- Need an *abstraction function* to map concrete states to abstract states.
  - For dictionary, map implementation to `<key,value>` pairs.
  - If the implementation is java.util.map, the contract for get(key) method:
    \[ (|<key, value> \in \emptyset(dict)|) \]
    \[ o = \text{dict.get}(k) \]
    \[ (|o = \text{value}|) \]
We Have Learned

- Symbolic execution is the process of establishing constraints on the values of variables as a particular path is taken.
  - Hand execution using symbols instead of concrete values. Rules governing any execution of a path.
  - Bridge from concrete execution of a complex program to mathematical logic structures that can be reasoned over.
  - Used to prove correctness of pieces of a program.
We Have Learned

- To perform over loops, methods, and data structures, must establish contracts (pre and post-conditions) on pieces of the program.
  - Can then reason about combinations of these pieces, as correctness is proven over the program hierarchy.
  - Allows checkable specifications of intended behavior.
Next Time

● Automated Test Case Generation

● Homework:
  ○ Reading assignment 3 - due April 10th.
  ○ Assignment 3 - due tonight!
  ○ Assignment 4 - out soon!