# Symbolic Execution and Proof of Properties



## Symbolic Execution

- Builds predicates that characterize
  - Conditions for executing paths
  - Effects of the execution on program state
- Bridges program behavior to logic
- Finds important applications in
  - program analysis
  - test data generation
  - formal verification (proofs) of program correctness



## Formal proof of properties

- Relevant application domains:
  - Rigorous proofs of properties of critical subsystems
    - Example: safety kernel of a medical device
  - Formal verification of critical properties particularly resistant to dynamic testing
    - Example: security properties
  - Formal verification of algorithm descriptions and logical designs
    - less complex than implementations



## Symbolic state

Values are expressions over symbols
Executing statements computes new expressions

| Execution with concrete values | | Execution with symbolic values | |
|---|---|---|---|
| before | | before | |
| low | 12 | low | L |
| high | 15 | high | H |
| mid | - | mid | - |
| mid = (high+low)/2 | | mid = (high+low)/2 | |
| after | | after | |
| low | 12 | Low | L |
| high | 15 | high | H |
| mid | 13 | mid | (L+H)/2 |

# Dealing with branching statements

## a sample program

Branching stmt

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



# Executing while (high >= low) {

Add an expression that records the condition for the execution of the branch (PATH CONDITION)

| | |
|---|---|
| | before |
| | low = 0 |
| and | high = (H-1)/2 -1 |
| and | mid = (H-1)/2 |

while (high >= low) {

| | |
|---|---|
| | after |
| | low = 0 |
| and | high = (H-1)/2 -1 |
| and | mid = (H-1)/2 |
| and | (H-1)/2 - 1 >= 0 |

if the TRUE branch was taken

... and not((H-1)/2 - 1 >= 0)

if the FALSE branch was taken



# Summary information

- Symbolic representation of paths may become extremely complex
- We can simplify the representation by replacing a complex condition P with a weaker condition W such that

  P => W
- W describes the path with less precision
- W is a *summary* of P



# Example of summary information

(Referring to Binary search: Line 17, *mid = (high+low)/2* )

- If we are reasoning about the correctness of the binary search algorithm, the complete condition:

  *low = L*  
  *and high = H*  
  *and mid = M*  
  *and M = (L+H)/2*
- Contains more information than needed and can be replaced with the weaker condition:

  *low = L*  
  *and high = H*  
  *and mid = M*  
  *and L <= M <= H*
- The weaker condition contains less information, but still enough to reason about correctness.

## Weaker preconditions

- The weaker predicate *L <= mid <= H* is chosen based on what must be true for the program to execute correctly
- It cannot be derived automatically from source code
- it depends 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**
- Weakening the predicate has a cost for testing:
  - satisfying the predicate is no longer sufficient to find data that forces program execution along that path.
    - test data that satisfies a weaker predicate W is necessary to execute the path, but it may not be sufficient
    - showing that W cannot be satisfied shows path infeasibility



## Loops and assertions

- The number of execution paths through a program with loops is potentially infinite
- To reason about program behavior in a loop, we can place within the loop an **invariant**:
  - assertion that states a predicate that is expected to be true each time execution reaches that point.
- Each time program execution reaches the invariant assertion, we can weaken the description of program state:
  - If predicate P represents the program state
  - and the assertion is W
  - we must first ascertain P => W
  - and then we can substitute W for P



## Pre- and post-conditions

- Suppose:
  - every loop contains an assertion
  - there is an assertion at the beginning of the program
  - a final assertion at the end
- Then:
  - every possible execution path would be a sequence of segments from one assertion to the next.
- Terminology:
  - Precondition: The assertion at the beginning of a segment,
  - Postcondition: The assertion at the end of the segment



## Verifying program correctness

- If for each program segment we can verify that
  - Starting from the precondition
  - Executing the program segment
  - The postcondition holds at the end of the segment
- Then
  - We verify the correctness of an infinite number of program paths

# Example

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

*Precondition*: is sorted:

```
Forall{i,j} 0 <= i < j < size :
dictKeys[i] <= dictKeys[j]
```

*Invariant*: in range

```
Forall{i} 0 <= i < size :
dictKeys[i] = key =>
low <= i <= high
```



# Executing the loop once...

Initial values:

```
low = L
and high = H
```

Precondition
Forall{i,j} 0 <= i < j < size
dictKeys[i] <= dictKeys[j]

Instantiated invariant:

```
Forall{i,j} 0 <= i < j < size :
dictKeys[i] <= dictKeys[j]
and Forall{k} 0 <= k < size :
dictKeys[k] = key => L <= k <= H
```

Invariant
Forall{i} 0 <= i < size :
dictKeys[i] = key =>
low <= i <= high

After executing: `mid = (high + low)/2`

```
low = L
and high = H
and mid = M
and Forall{i,j} 0 <= i < j < size :
dictKeys[i] <= dictKeys[j]
and Forall{k} 0 <= k < size :
dictKeys[k] = key => L <= k <= H
and H >= M >= L
```

....



# ...executing the loop once

After executing the loop

```
low = M+1
and high = H
and mid = M
and Forall{i,j} 0 <= i < j < size :
dictKeys[i] <= dictKeys[j]
and Forall{k} 0 <= k < size :
dictKeys[k] = key => L <= k <= H
and H >= M >= L
and dictkeys[M]<key
```

The new instance of the invariant:

```
Forall{i,j} 0 <= i < j < size :
dictKeys[i] <= dictKeys[j]
and Forall{k} 0 <= k < size :
dictKeys[k] = key => M+1 <= k <= H
```

If the invariant is satisfied, the loop is correct wrt the preconditions and the invariant



# From the loop to the end

If the invariant is satisfied, but the condition is false:

```
low = L
and high = H
and Forall{i,j} 0 <= i < j < size :
dictKeys[i] <= dictKeys[j]
and Forall{k} 0 <= k < size :
dictKeys[k] = key => L <= k <= H
and L > H
```

If the the condition satisfies the post-condition, the program is correct wrt the pre- and post-condition:

# Compositional reasoning

- Follow the hierarchical structure of a program
  - at a small scale (within a single procedure)
  - at larger scales (across multiple procedures…)
- Hoare triple: [pre] block [post]
- 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*



# Reasoning about Hoare triples: inference

premise

$$\frac{\text{[I and C] S [I]}}{\text{[I] while(C)\{S\} [I and notC]}}$$

conclusion

Inference rule says:
 if we can verify the premise (top),
 then we can infer the conclusion (bottom)



# Some other rules: if statement

$$\frac{\text{[P and C] thenpart [Q]} \qquad \text{[P and notC] elsepart [Q]}}{\text{[P] if (C)\{thenpart\} else \{elsepart\} [Q]}}$$



# Reasoning style

- Summarize the effect of a block of program code (a whole procedure) by a *contract* == precondition + postcondition
- Then use the contract wherever the procedure is called

*example*

summarizing **binarySearch**:

 (forall i,j, 0 <= i < j < size : keys[i] <= keys[j])

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

 (s=v and exists i , 0 <= i , size : keys[i] = k and vals[i] = v)

or

 (s=v and not exists i , 0 <= i , size : keys[i] = k)

## Reasoning about data structures and classes

- Data structure module = collection of procedures (methods) whose specifications are strongly interrelated
- Contracts: specified by relating procedures to an abstract model of their (encapsulated) inner state

*example:*

Dictionary can be abstracted as {<key, value>}

independent of the implementation as a list, tree, hash table, etc.




## Structural invariants

- Structural characteristics that must be maintained as specified as *structural invariants* (~loop invariants)
- Reasoning about data structures
  - if the structural invariant holds before execution
  - and each method execution preserve the invariant
  - …then the invariant holds for all executions

*Example:* Each method in a search tree class maintains the ordering of keys in the tree




## Abstraction function

- maps concrete objects to abstract model states

Dictionary example

abstraction function

[<k,v> in Φ(dict) ]

o = dict.get(k)

[ o = v ]




## Summary

- Symbolic execution = bridge from an operational view of program execution to logical and mathematical statements.
- Basic symbolic execution technique: execute using symbols
- Symbolic execution for loops, procedure calls, and data structures: proceed hierarchically
  - compose facts about small parts into facts about larger parts
- Fundamental technique for
  - Generating test data
  - Verifying systems
  - Performing or checking program transformations
- Tools are essential to scale up