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

<table>
<thead>
<tr>
<th>Execution with concrete values</th>
<th>Execution with symbolic values</th>
</tr>
</thead>
<tbody>
<tr>
<td>before</td>
<td>before</td>
</tr>
<tr>
<td>low 12</td>
<td>low L</td>
</tr>
<tr>
<td>high 15</td>
<td>high H</td>
</tr>
<tr>
<td>mid -</td>
<td>mid -</td>
</tr>
<tr>
<td>mid = (high+low)/2</td>
<td>mid = (high+low)/2</td>
</tr>
<tr>
<td>after</td>
<td>after</td>
</tr>
<tr>
<td>low 12</td>
<td>Low L</td>
</tr>
<tr>
<td>high 15</td>
<td>high H</td>
</tr>
<tr>
<td>mid 13</td>
<td>mid (L+H)/2</td>
</tr>
</tbody>
</table>
Dealing with branching statements

A sample program

```c
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)

```c
before
low = 0
and high = (H-1)/2 - 1
and mid = (H-1)/2
while (high >= low) {
    if (comparison < 0) {
        low = mid + 1;
    } else if (comparison > 0) {
        high = mid - 1;
    } else {
        return dictValues[mid];
    }
}
```

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

if the FALSE branch was taken

if the TRUE branch was taken

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.

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
Weaker preconditions

- The weaker predicate $L \leq mid \leq H$ 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

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

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 $P \Rightarrow W$
  - and then we can substitute $W$ for $P$

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[], 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 the loop once...

Initial values:
  low = L
  high = H

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

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

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

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: \([\text{pre}] \text{ block } [\text{post}]\)

- if the program is in a state satisfying the precondition \(\text{pre}\) at entry to the block, then after execution of the block it will be in a state satisfying the postcondition \(\text{post}\)

Reasoning about Hoare triples: inference

<table>
<thead>
<tr>
<th>premise</th>
<th>([I \text{ and } C] \text{ S } [I])</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>([I] \text{ while}(C){S} [I \text{ and not } C])</td>
</tr>
</tbody>
</table>

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

Some other rules: if statement

\[
\begin{align*}
[P \text{ and } C] \text{ thenpart } [Q] & \quad [P \text{ and not } C] \text{ elsepart } [Q] \\
[P] \text{ if } (C)\{\text{thenpart}\} \text{ else } \{\text{elsepart}\} [Q]
\end{align*}
\]

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

\textbf{example}

summarizing \texttt{binarySearch}:

\(\forall i, j, 0 \leq i < j < \text{size} : \texttt{keys[i]} \leq \texttt{keys[j]}\)
\(s = \texttt{binarySearch}(k, \texttt{keys}, \texttt{vals}, \text{size})\)
\(s = v \text{ and exists } i, 0 \leq i, \text{size} : \texttt{keys[i]} = k \text{ and } \texttt{vals[i]} = v\)
or
\(s = v \text{ and not exists } i, 0 \leq i, \text{size} : \texttt{keys[i]} = k\)
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

\[
\begin{align*}
\text{abstraction function} \\
[k,v] \in \Phi(\text{dict}) \\
o = \text{dict.get}(k) \\
[ o = v ]
\end{align*}
\]

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