Verification: Part 2: Analysis of design artifacts

- Informal and formal techniques
- Symbolic execution
- Use of analysis to complement testing
- GJZ sections 6.4 and 6.5

Recall

Verification problem: is to check that an artifact is implemented correctly

Two general approaches:
- execute the artifact (testing)
- prove properties of the artifact itself (analysis)

Approaches have different strengths and weaknesses and are complementary

Analysis (for verification)

Distinct from “problem analysis"
Characterizes a class of executions
May be formal or informal

Formal techniques based on a model
- e.g., a formal specification of the program
- e.g., an artifact that is systematically derived from the program

Informal: Code walkthrough

Group meeting to analyze the code of a system
Recommended prescriptions
- Small number of people (three to five)
- Participants receive written documentation from the designer a few days before the meeting
- Predefined duration of meeting (a few hours)
- Focus on the discovery of errors, not on fixing them
- Participants: designer, moderator, and a secretary
- Foster cooperation; no evaluation of people
  - Experience shows that most errors are discovered by the designer during the presentation, while trying to explain the design to other people.

Informal: Code inspection

Reading technique aiming at error discovery
Based on checklists; e.g.:
- use of uninitialized variables;
- jumps into loops;
- non-terminating loops;
- array indices out of bounds;
- ...

Formal techniques

Systematically check or derive properties (or compute residual programs) from a program
- Symbolic execution
- Abstract interpretation
- Model checking
- Partial evaluation
- Program slicing

Many have become practical only recently
Motivation for techniques

Simplest model of the "state" of a program in execution is a 4-tuple of the form: $<\text{store, pc, infile, outfile}>$

Where:
- store is a set of bindings of variables to values
  - e.g., $\{x=2.3, y=50, s=\text{"Hello world"}\}$
- pc is the program counter
  - records the location of control in the program
- infile and outfile represent input and output files respectively

Example

Suppose program is executed with the input file: $<4, 10>$

Example

Initial state: $<\{i=1, a=1\}, 1, <4, 10>, <>>$

Executing first edge yields: $<\{i=4, a=10\}, 2, <>, <>>$

Executing next edge yields: $<\{i=5, a=13\}, 4, <>, <>>$

Example

Executing first edge yields: $<\{i=4, a=10\}, 2, <>, <>>$

Executing next edge yields: $<\{i=5, a=13\}, 4, <>, <>>$
Example

Executing next edge yields:
\(< {i=5, a=13},
2,\)
\(<>,\)
\(< > \)

\(\alpha := \alpha + 3;\)
\(i := i + 1;\)

\(\text{write} (\alpha);\)

FALSE

TRUE

Example

Executing next edge yields:
\(< {i=5, a=13},
5,\)
\(<>,\)
\(< > \)

\(\alpha := \alpha + 3;\)
\(i := i + 1;\)

\(\text{write} (\alpha);\)

FALSE

TRUE

Example

Finally:
\(< {i=5, a=13},
6,\)
\(<>,\)
\(<13> > \)

\(\alpha := \alpha + 3;\)
\(i := i + 1;\)

\(\text{write} (\alpha);\)

Symbolic execution

Program executed using symbolic values
– e.g., variables bound to values of the form \(A\) or \(A+3\) rather than 10 or 13

“State of execution” records:
– binding of variables to symbolic values,
– representation of the path executed to reach the state,
– a path condition, which accumulates constraints on symbolic values required to force program to exercise the path

Symbolic execution (continued)

More formally, the symbolic state is a triple of the form:
\(<\text{symbolic binding}, \text{path}, \text{path condition}>\)

One symbolic execution corresponds to many actual executions
Useful for determining conditions sufficient to exercise a given path

Application: Selecting paths in testing

Coverage criteria often dictate a profile for choosing paths of execution to test
– E.g., want to execute loops at their boundaries (0, 1, \(n-1, n\)).
– Easy to name a path to test

Problems:
– Not trivial to derive test data to execute paths
– Not every path in a CFG corresponds to a feasible execution of the program
Symbolic execution and testing

The path condition describes the data that traverse a certain path

Use in testing:
– select path
– symbolically execute it
– synthesize data that satisfy the path condition
  • they will execute that path

Example

Suppose we want to execute the path: <1,2,5,6>

Initial symbolic state:
< {i=I, a=A, inC=0, outC=0}, <1>, true >

Execute first edge:
< {i=in[1], a=in[2], inC=2, outC=0}, <1,2>, true >

Execute next edge:
< {i=in[1], a=in[2], out[1] = in[2], inC=2, outC=1}, <1,2,5>, true >

Execute final edge:
< {i=in[1], a=in[2], out[1] = in[2], inC=2, outC=2}, <1,2,5,6>, true >
Use path condition to derive inputs

Path condition helps to identify inputs that cause program to exercise desired path.
Path condition:
\[ \text{in}[1] \geq 5 \land \text{inC}=2 \]
Tells us we need an input file with:
- Two elements
- First of which is a number greater than or equal to 5
Easy to choose such a data set.

Symbolic execution rules:

Case 1: source not a decision

\[
\begin{align*}
\{ sb \} \text{ e } \{ sb \} & \quad \text{last}(p) = s \\
<sb, p, pcond> & \quad <sb', p, <t>, pcond>
\end{align*}
\]

Symbolic execution rules:

Case 2a: evaluable decision

\[
\begin{align*}
\text{eval}(\text{cond}(s), sb) &= \text{TRUE} & \text{last}(p) &= s \\
<sb, p, pcond> & \quad <sb, p, \text{true}, pcond>
\end{align*}
\]

\[
\begin{align*}
\text{eval}(\text{cond}(s), sb) &= \text{FALSE} & \text{last}(p) &= s \\
<sb, p, pcond> & \quad <sb, p, \text{false}, pcond>
\end{align*}
\]

Symbolic execution rules:

Case 2b: non-evaluable decision

\[
\begin{align*}
\text{eval}(\text{cond}(s), sb) &= \bot & \text{last}(p) &= s \\
<sb, p, pcond> & \quad <sb, p, \text{false}, pcond, \neg \text{simp}(\text{cond}(s), sb)>
\end{align*}
\]

Edge rules

\[
\begin{align*}
\{ sb \} \text{ read}(x) & \quad \{ sb [\text{inC} \leftarrow \text{inC}+1, \\
\quad x \leftarrow \text{in[\text{inC}+1]}] \} \\
\{ sb \} \text{ write}(E) & \quad \{ sb [\text{outC} \leftarrow \text{outC}+1] \} \\
\{ sb \} x := E & \quad \{ sb [x \leftarrow \text{simp}(E, sb)] \}
\end{align*}
\]
Example
Suppose we want to execute the path: \(<1,2,3,4,2,3,4,2,5,6>\)

Example
Initial symbolic state:
\(<\{i=I, a=A, \text{inC}=0, \text{outC}=0\},
<1>,
true \rangle \)

Example
Execute first edge:
\(<\{i=\text{in}[1], a=\text{in}[2], \text{inC}=2, \text{outC}=0\},
<1,2>,
true \rangle \)

Example
Execute next edge:
\(<\{i=\text{in}[1] + 1, a=\text{in}[2] + 3,
\text{inC}=2, \text{outC}=0\},
<1,2,3>,
in[1] < 5 \rangle \)

Example
Execute next edge:
\(<\{i=\text{in}[1] + 1, a=\text{in}[2] + 3,
\text{inC}=2, \text{outC}=0\},
<1,2,3,4>,
in[1] < 5 \rangle \)
Example

Execute next edge:
\(< \{ i = in[1] + 1, a = in[2] + 3, inC = 2, outC = 0 \},
\langle 1, 2, 3, 4, 2, 3, 4, 2 \rangle,\in[1] < 5 \land \in[1] + 1 < 5 >\)
\(a := a + 3; i := i + 1;\)

Example

Execute next edge:
\(< \{ i = in[1] + 1, a = in[2] + 3, inC = 2, outC = 0 \},
\langle 1, 2, 3, 4, 2, 3, 4, 2, 5 \rangle,\in[1] < 5 \land \in[1] + 1 < 5 \land \in[1] + 2 \geq 5 \land inC = 2 >\)
\(a := a + 3; i := i + 1;\)

Example

Execute final edge:
\langle 1, 2, 3, 4, 2, 3, 4, 2, 5, 6 \rangle,\in[1] < 5 \land \in[1] + 1 < 5 \land \in[1] + 2 \geq 5 \land inC = 2 >\)
\(a := a + 3; i := i + 1;\)

Now derive inputs from path cond

Path condition:
\(\in[1] < 5 \land \in[1] + 1 < 5 \land \in[1] + 2 \geq 5 \land inC = 2\)

Tells us we need an input file with:
- Two elements
- First of which is a number less than four and greater than or equal to 3

Easy to choose such a data set!
Problem of infeasible paths

A feasible path is one for which there exist inputs that cause the program to traverse the path.

Easy to choose infeasible paths when trying to satisfy some coverage criterion:
- Such paths must be replaced with feasible “equivalents” to satisfy coverage.
- Symbolic execution also useful for identifying infeasible paths!

Example

Suppose we want to execute the path: <1,2,3,4,2,5,6>

Example (continued)

Symbolic execution of the desired path yields the following path condition:
\[ \text{in}[2] < \text{in}[2] \land \text{in}[2]+3 \geq \text{in}[2] + 3 \land \text{inC}=2 \]

Notice that this condition is not satisfiable.

In general, one would need a theorem prover to determine satisfiability/solution to a path condition.