Analysis Overview


Basic Definitions

- Failure?
- Fault/defect?
- Error?

![Diagram showing concepts of intent, observed failure, comparison, model/product, behavior, and fault]
Define

- Validation and Verification: V&V
- Correctness
- Reliability
- Robustness
- Conservative analysis

Formal models

- Analysis is usually done on a model of an artifact
  - textual representation of the artifact is translated into a model that is more amenable to analysis than the original representation
  - the translation may require syntactic and semantic analysis so that the model is as accurate as possible
  - Conservative means?
- graphs are the most common forms of models use
- Ideally want general models
  - different languages e.g., Ada, C++, Java
  - different levels of abstraction/detail e.g., detailed design, arch. design
  - different kinds of artifacts e.g., code, designs, requirements
Approaches

- **Static Analysis**
  - Inspections
  - Software metrics
  - Symbolic execution
  - Dependence Analysis
  - Data flow analysis
  - Software Verification

- **Dynamic Analysis**
  - Assertions
  - Error seeding, mutation testing
  - Coverage criteria
  - Fault-based testing
  - Specification-based testing
  - Object-oriented testing
  - Regression testing

Reviews, Inspections, and Walkthroughs

- Manual static analysis methods
- Most can be applied at any step in the lifecycle
- Have been shown to improve reliability, but
  - often the first thing dropped when time is tight
  - labor intensive
  - often done informally, no data/history, not repeatable
Reviews in the RUP

Reviews, Inspections, and Walkthroughs

- **Formal reviews**
  - author or one reviewer leads a presentation of the product
  - review is driven by presentation, issues raised

- **Walkthroughs**
  - usually informal reviews of source code
  - step-by-step, line-by-line review

- **Inspections**
  - list of criteria drive review
  - properties not limited to error correction
  - historical context
Review methods

- Fagan inspections
  - formal, multi-stage process
  - significant background & preparation
  - led by moderator
- Active design reviews
  - also called “phased inspections”
  - several brief reviews rather than one large review
  - guided by questions from the author
- Cleanroom
  - more than reviews, but reviews important component
  - we'll come back to this
- N-fold
  - parallel reviews controlled by moderator
  - focuses on user requirements

Fagan Inspections (3-5 participants)

- Moderator
  - Responsible for organizing, scheduling, distributing materials, and leading the session
- Author
  - Responsible for explaining the product
- Scribe
  - Responsible for recording bugs found
- Planner or designer
  - Author from a previous step in the software lifecycle
- User representative
  - To relate the product to what the user wants
- Peers of the author
  - Perhaps more experienced, perhaps less
- Apprentice
  - An observer who is there mostly to learn
Fagan Inspection Process

- **Planning**
  - Gather materials and insure that they meet entry criteria
  - Arrange for participants, assign them roles, insure their training
  - Arrange meeting
- **Overview**
  - Explain content to the inspectors
- **Preparation**
  - Participants study material
- **Inspection**
  - Find/Report faults (Do not discuss alternative solutions)
- **Rework**
  - Author fixes all faults
- **Follow-Up**
  - Team certifies faults fixed and no new faults introduced

Fagan Inspection

- **General guidelines**
  - Distribute material ahead of time
  - Use a written checklist of what should be considered
    - e.g., functional testing guidelines
  - Criticize product, not the author
Experimental Results

- using software inspections has repeatedly been shown to be cost effective
- increases front-end costs
  - ~15% increase to development cost
- decreases overall cost

IBM study
- doubled number of lines of code produced per person
  - some of this due to inspection process
- reduced faults by 2/3
- found 60-90% of the faults
- found faults close to when they are introduced
  - helps reduce cost

People Resource vs. Schedule

- Diagram showing people resource vs. schedule with and without inspections.
Why are inspections effective

- knowing the product will be scrutinized causes developers to produce a better product
- having others scrutinize a product increases the probability that faults will be found
- walkthroughs and reviews are not as formal as inspections, but appear to also be effective
  - hard to get empirical results

What are the deficiencies?

- focus on error detection
  - what about other "ilities" -- maintenance, portability, etc.
- not applied consistently & rigorously
  - inspection shows statistical improvement, but cannot ensure quality
  - inspection should have the same results without regard to the product to which it is applied or the inspection team
- range of errors not addressed
  - team expertise limited
  - one property may have many error modalities
- human intensive and often makes ineffective use of human resources
  - e.g., skilled software engineer reviewing coding standards, comments spelling, etc.
- no automated support
  - again inefficient of human resources
- aspects of review not used appropriately
  - e.g., in Fagan process, overview often covers what should be described if documentation is adequate
Cleanroom

Processes

- Customer Requirements
- Specification
- Implementation
- Usage Specification
- Incremental Development Planning
- Statistical Testing
- Quality Certification Model

Work Products

- Customer Requirements
- Specification
- Usage Specification
- Incremental Development Plan
- Test Cases
- Failure Data
- Improvement Feedback

Measures of Operational Performance

- Customer Requirements
- Specification
- Usage Specification
- Incremental Development Planning
- Statistical Testing
- Quality Certification Model

Cleanroom

- Verification as Review Process
  - Team verification of correctness takes the place of individual unit testing; correctness is established by group consensus if it is obvious.
  - by formal proof techniques if it is not.
  - Intellectual control of the process
  - motivates developers to deliver error-free code
  - verification is a form of peer review
  - each person assumes responsibility for and derives a sense of ownership in the evolving product
  - every person must agree that the work is correct before it is accepted -> successes are ultimately team successes, and failures are team failures.
  - Markov Analysis
  - Factors
    - number of statistically typical (i.e., likely) usage paths through the software
  - Steps
    - focus verification efforts,
    - identify the likelihood of given events,
    - project the test schedule, and
    - ascertain the (affordable) upper bound on inferences about reliability
  - Stopping Criterion for Testing
    - goals (e.g., target level of estimated reliability) are achieved
    - or quality standards (e.g., errors/KLOC) are violated
Generation of Test Cases

- usage model -> test cases
- may be automatically generated.
- each test case is a random walk through the usage model
- invocation -> termination
- test cases constitute a "script" for use in testing
- may be applied by human testers, or used as input to an automated test tool.

Stopping Criterion for Testing

- goals (e.g., target level of estimated reliability) are achieved
- or quality standards (e.g., errors/KLOC) are violated

Statistical Hypothesis Testing

<table>
<thead>
<tr>
<th>Reliability level (r)</th>
<th>90</th>
<th>95</th>
<th>99</th>
<th>99.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>22</td>
<td>29</td>
<td>44</td>
<td>66</td>
</tr>
<tr>
<td>0.95</td>
<td>45</td>
<td>59</td>
<td>90</td>
<td>135</td>
</tr>
<tr>
<td>0.99</td>
<td>230</td>
<td>299</td>
<td>459</td>
<td>688</td>
</tr>
<tr>
<td>0.999</td>
<td>2302</td>
<td>2993</td>
<td>4603</td>
<td>6905</td>
</tr>
</tbody>
</table>

Software Metrics

- measures that predict qualities about software
- can be applied to any of the products (e.g., design, code, test cases) or to the process (e.g., Capability Maturity Model)

Qualities measured by software metrics

- performance
- user-friendliness
- resources
- memory/storage
- development costs
- maintenance cost
- quality
- maintainability
- reliability
- completeness
- consistency
- complexity
Function Points

- proposed by Albrecht in 1979
- Originally applied to code
- UFP =
  - number of inputs x w1 +
  - number of outputs x w2 +
  - number of user inquiries x w3 +
  - number of files x w4 +
  - number of external references x w5
- function points = UFP* TCA = UFP* (.65 + 0.01 * SUM(Fi))
- where the degree of influence, DI= SUM(Fi) is the sum of complexity adjustment values, Fi

metrics:
- productivity: FP/person-month
- quality: defects/FP
- cost: $/FP

weights:
<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>Average</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>w2</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>w3</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>w4</td>
<td>7</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>w5</td>
<td>5</td>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>

McCabe’s cyclomatic complexity

- Complexity measured by control flow information
- based on a control flow graph where e is number of edges, n is number of nodes, p is number of connected components
- McCabe’s Cyclomatic Complexity:
  - \( v = e - n + 2 \)
  - where:
    - \( v \) = complexity of the graph
    - \( e \) = number of edges (program flows between nodes)
    - \( n \) = number of nodes (sequential groups of program statements)
- if a strongly connected graph is constructed (one in which there is an edge between the exit node and entry node), the calculation is
  - \( v = e - n + 1 \)
Example

\[ C = 10 - 8 + 2 = 4 \]

Quality Metrics for Code

- Understandability
  - size metrics
    - lines of code
    - function points
    - function count
  - traceability metrics
    - number of comment lines per total source lines of code
    - percent comment lines of total lines
    - correctness of comments
- Predicting quality
  - LOC X domain seems to be the most reliable predictor
## Analysis

**Intent**

- **Testing**
  - Dynamic Analysis
  - Static Analysis

**Comparison**

- **Behavior**
  - inferred
  - observed

**Model/Product**

## Approaches

### Static Analysis
- Inspections
- Software metrics
- Symbolic execution
- Dependence Analysis
- Data flow analysis
- Software Verification

### Dynamic Analysis
- Assertions
- Error seeding, mutation testing
- Coverage criteria
- Fault-based testing
- Specification-based testing
- Object-oriented testing
- Regression testing
Basic Verification Strategy

- analyze a system for desired properties, i.e., compare behavior to intent
  - Intent can be expressed as?

- behavior can be?

Basic Verification Strategy

- different representations support different sorts of inferences

- comparison can be informal
  - done by human eye, e.g., inspection

- can be done by computers
  - comparing text strings

- can be done by model-checkers
  - such as formal machines (e.g., fsa's)

- can be done by rigorous mathematical reasoning
Example: Dataflow Analysis

- **intent:**
  - stated as a property
  - captured as an event sequence

- **behavior:**
  - model represents some execution characteristics
  - inferred from a model: (e.g., annotated flow graph)
  - inferences based upon:
    - semantics of flow graph
    - semantics captured by annotations

- **comparison:**
  - done by a fsa (e.g., a property automaton)

Data Flow Analysis

"property" = Cecil constraint

**Intent**

if dfa accepts all traces then the constraint holds for all computations

**Comparison**

open

dfa defined by Cecil constraint
close

trace = computation along path in an annotated dataflow graph

model
Data Flow Analysis (DFA)

- Uses an annotated control flow graph model of the program
  - Compute facts for each node
  - Use the flow in the graph to compute facts about the whole program
- DFA used extensively in program optimization, e.g.,
  - determine if a definition is dead (and can be removed)
  - determine if a variable always has a constant value
  - determine if an assertion is always true and can be removed
- Some Dataflow systems
  - DAVE system demonstrated the ability to find def/ref anomalies
  - Cecil/Cesar system demonstrated the ability to prove general user-specified properties
  - FLAVERS demonstrated applicability to concurrent system

Data flow analysis

- Computes information that is true at each node in the CFG, e.g.,
  - what variables are defined
  - what variables are referenced
- Usually stored in sets
  - ref(n) is the set of variables referenced at node n
- Uses this local information and the control flow graph to compute global information about the whole program
  - Done incrementally by looking at each node’s successors or predecessors
Def-ref path expressions

- For a path $P$ and a variable $\alpha$, can write a path expression describing the sequence of set memberships encountered for $\alpha$, where
  - $\alpha \in \text{def}(n)$ or
  - $\alpha \in \text{ref}(n)$ or
  - $\alpha \in \text{null}(n)$
- For each node $n$ on the path
  - Write (and simplify) a path expression

$P(n_1, n_2, ..., n_k; \alpha) = d11r = dr$
### Anomalous pairs of ref/defs

<table>
<thead>
<tr>
<th></th>
<th>d - defined, r - referenced, u - undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd</td>
<td>bug?</td>
</tr>
<tr>
<td>dr</td>
<td>normal</td>
</tr>
<tr>
<td>uu</td>
<td>harmless?</td>
</tr>
<tr>
<td>rd</td>
<td>normal</td>
</tr>
<tr>
<td>ur</td>
<td>bug</td>
</tr>
</tbody>
</table>

#### Consider unreferenced definition

- **Want to know if a def is not going to be referenced**
  - dd or du

- **At the point of a definition of a, want to know if there is some path where a is defined or undefined before being used**
  - May be indicative of a problem if the path is executable
  - Usually just a programming convenience and not a problem

- **At the point of a definition of a, want to know if on all paths a is defined or undefined before being used**
  - May be indicative of a problem
  - Or could just be wasteful
global dataflow analysis

- classes
  - forward flow problems (e.g., available expressions)
    - what definitions can affect computations at a given point in a program
  - backward flow problems (e.g., live variables)
    - what uses that follow a given point in the program can be affected by computations up to that point

- paths
  - any path
  - all path

Unreferenced definitions

int x, y;
...
x := 3;
y := x + 2;
if x > 0 then
  x := x + y;
end if;
y := ...

Forward flow, all paths problem

(int unreferenced defs)

Need to look at each node where there is a def
General Approach

- Initial values
  - for each node define gen and kill information

- Input Equations
  - for each node we have an equation of the form: $\text{In}_i := \text{Merge} (\text{Out}_j)$
  - “Merge” operation over the “predecessors” of $n_i$

- Input Equations
  - for each node we have an equation of the form: $\text{Out}_i := \text{Merge} (\text{In}_i)$

- Transfer Equations
  - for each node we have an equation of the form: $\text{Out}_i := f_i(\text{In}_i)$
  - Transfer functions usually depend on Gen/Kill information that is computed for each node
  - Usually: $\text{Out} := (\text{In} - \text{kill})U \text{gen}$
  - We can view the set of variables, transfer functions, and flow graph as a system of equations
**worklist algorithm**

1. Start at initial node (entry for forward; exit for reverse), label \( \text{IN}_0 \) with pertinent "facts" (initial values)
2. Compute \( \text{OUT}_0 = F(\text{IN}_0) \) (label \( \text{OUT}_0 \) with the computed facts)
3. Propagate \( \text{OUT}_0 \) to \( \text{IN} \) (label edge \( N_i \Rightarrow \text{N} \) with \( \text{OUT}_0 \)) where \( N_i \) are successor nodes (forward) or predecessor nodes (reverse) of \( N_0 \)
4. Compute \( \text{OUT}_i = F(\text{IN}_i) \), place all \( N_i \) on a "worklist" \( W \), and for all \( N_i \) label \( \text{OUT}_i \) with the computed facts.
5. While \( W \) is not empty,
   1. pick \( N_i \) from \( W \) and propagate \( \text{OUT}_i \) to \( \text{IN}_i \) (label edges \( N_i \Rightarrow \text{N} \) with \( \text{OUT}_i \)) where \( N_i \) are successor nodes (forward) or predecessor nodes (reverse) for \( N_0 \); delete \( N_i \) from \( W \)
   2. Compute \( \text{OUT}_i = F(\text{IN}_i) \) for all \( N_i \) where \( \text{IN}_i = \text{MERGE} \) all input edge labels (\( \text{MERGE} = \cup \) for "some paths" and \( \cap \) for "all paths"), label \( \text{OUT}_i \) with the computed facts); and if for \( N_i \), \( \text{OUT}_i \) changes put \( N_i \) on \( W \)
6. If \( W \) is not empty, then \( W = W' \) and go to 5

---

**Using Quantified Regular Expressions**

- Alphabet, quantification, regular expression
- For the events \{open, close, move\}
  show that for all paths:
    \((\text{close} \lor \text{move})^*, \ (\text{open}^* \lor \text{open}^* \lor \text{close})^*\)
Cecil: Olender and Osterweil

- Instead of implicitly defined facts, let the user define application-specific facts
  - Represented as a Deterministic Finite State Automaton (DFSA) or as a Quantified Regular Expression (QRE)
- Events
  - Recognizable events
  - Method calls
    - Can reason about sequences of method calls
      - E.g., Push must be called before Pop
  - Thread interactions
    - Join or Fork
  - Arbitrary operations
    - a+b
  - Need to be able to treat events as indivisible actions
    - E.g., can treat pop and push as atomic as long as they do not contain any events of concern
- Propagate the states in the DFSA that can reach each node in the program

State Propagation

- States of the property are propagated through the CFG
- The property is proved if only accepting (non-accepting) states are contained in the final node of the CFG
- Cecil DFSA -
  - lattice ($\hat{P}(S)$, $\subset$, $\cup$)
  - function space
    - $\delta : \hat{P}(S) \rightarrow \hat{P}(S)$
    - facts at nodes are elements of $\hat{P}(S)$
  - propagate until convergence and check if terminal node in an accepting state of DFSA
Elevator Controller

```c
void main()
{
    1: if (elevatorStopped)
        {...
    3: openDoors();
        }
    5: if (elevatorStopped)
        {...
    7: closeDoors();
        }
    9: moveToNextFloor();
}
```

- States of the property are propagated through the CFG
- For an all property: the property is proved if only accepting states are contained in the final node of the CFG
- For a none property: the property is proved if only non-accepting states are contained in the final node of the CFG

**State propagation**

![State propagation diagram]
### Approaches

- **Static Analysis**
  - Inspections
  - Software metrics
  - Symbolic execution
  - Dependence Analysis
  - Data flow analysis
  - Software Verification

- **Dynamic Analysis**
  - Assertions
  - Error seeding, mutation testing
  - Coverage criteria
  - Fault-based testing
  - Specification-based testing
  - Object-oriented testing
  - Regression testing
Verification

- two well-established approaches
  - (Automated) mathematical reasoning
    - theorem proving
    - proof checking
  - Finite-state verification
    - model checking
      - Logic spec + FSA comp model $\Rightarrow$ symbolic model checking
      - FSA spec + FSA comp model $\Rightarrow$ automata-theoretic model checking
    - property checking

How are they different?

- (Automated) mathematical reasoning
  - difficult, error prone
  - decidability vs. expressiveness
    - Propositional calculus is decidable
    - Predicate calculus is semi-decidable
- Finite-state verification
  - Reason about a finite model of the system
  - Fast, yields counterexamples, manages partial specifications, applies to concurrency
  - State explosion!


---

**Behaviors**

- Model/Product
- Intent
- Lemmas and theorems in predicate logic
- Typically inferred by symbolic execution of the specifications

---

**Floyd Method of Inductive Assertions**

- Show that given the input assertions, after executing the program, program satisfies output assertions
  - Show that each program fragment behaves as intended
  - Use induction to prove that all fragments, including loops, behave as intended
- Show that the program must terminate
- Informal description
  - Place assertions at the start, final, and intermediate points in the code.
  - Any path is composed of sequences of program fragments that start with an assertion, are followed by some assertion free code, and end with an assertion
    - $A_s, C_1, A_2, C_2, A_3, \ldots, A_{n-1}, C_{n-1}, A_f$
  - Show that for every executable path, if $A_s$ is assumed true and the code is executed, then $A_f$ is true
Why does this work?

- suppose P is an arbitrary path through the program
- can denote it by
  \[ P = A_0 \ C_1 \ A_1 \ C_2 \ A_2 \ldots \ C_n \ A_n \]
- where
  \[ A_0 \ - \ Initial \ assertion \]
  \[ A_n \ - \ Final \ assertion \]
  \[ A_i \ - \ Intermediate \ assertions \]
  \[ C_i \ - \ Loop \ free, \ uninterrupted, \ straight-line \ code \]

If it has been shown that
\[ \forall \ i, \ 1 \leq i < n: \ A_i C_i \Rightarrow A_{i+1} \]
Then, by transitivity
\[ A_0 \Rightarrow \ldots \Rightarrow A_n \]

Obvious problems

- How do we do this for a path?
- How do we do this for all paths?
  - Infinite number of paths
    - Must find a way to deal with loops
Find loop invariant \(A_i\)

- subpaths to consider:
  - \(C_1\) Initial assertion \(A_0\) to final assertion \(A_f\)
  - \(C_2\) Initial assertion \(A_0\) to \(A_i\)
  - \(C_3\) \(A_i\) to \(A_i\)
  - \(C_4\) \(A_i\) to final assertion \(A_f\)

- Basically an inductive proof

The “Aha!” moment - finding invariants is hard!

Wensley’s Algorithm

Procedure Wensley (P:input, Q:input, E:input, Y:output);
Declare P, Q, E, Y, A, B, D real;
A := 0.0;
B := Q/2.0;
D := 1.0;
Y := 0.0;
Do_While (D>=E)
  If \(\neg(P - A - B \geq 0.0)\) then
    \{ Y := Y+(D/2.0); A := A+B; \}
  B := B/2.0;
  D := D/2.0;
End_do;
End Wensley;
Floyd Proof: Wensley's Algorithm

- Summary of Five Lemmas Needed
  - $A_0$ to $A_i$
  - $A_i$, true branch, to $A_i$
  - $A_i$, false branch, to $A_i$
  - $A_i$, true branch, to $A_F$
  - $A_i$, false branch, to $A_F$

- Floyd Proof:

```
A: \{(A=Q*Y) \land (B=Q*(D/2)) \land (k \geq 0, k \text{ integer } \land D=2^k) \land ((P/Q)-D)<Y \leq (P/Q)\}
```

- Lemma III: $A_i$, false branch, to $A_i$

```
A': \{(A'=Q*Y') \land (B'=Q*(D'/2)) \land (k \geq 0, k \text{ integer } \land D'=2^k) \land ((P/Q)-D')<Y' \leq (P/Q)\}
```

- Code

```
Y ← Y+(D/2.0)
A ← A+B
B ← B/2
D  D/2
```
**proof of lemma III**

\[ A_1 \Rightarrow A'_1; \ ((A' = Q^* Y') \land (B' = Q^* (D'/2)) \]
\[ \land (k \geq 0, k \text{ integer} \land D' = 2^{-k}) \]
\[ \land ((P/Q) - D') < Y' \leq (P/Q) \]

we have

\[ A' = A + B; \ B' = B/2.0; \ D' = D/2.0; \ Y' = Y + D/2.0; \]

1) \[ A' = A + B = Q^* Y + Q^* (D/2) = Q^* (Y + (D/2)); \]
\[ Y' = Y + (D/2); \therefore A' = Q^* Y' \]

2) \[ B' = B/2 = (Q^* D/2)/2; \ D' = D/2 \]
\[ \therefore B' = (Q^* 2D/2)/2 = Q^* D/2 \]

and so on … basically using symbolic evaluation