Verification

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

predicate logic assertions

Intent

lemmas and theorems in predicate logic

typically inferred by symbolic execution of the specifications

model/product

Behavior

Static Analysis

Floyd Inductive Proof

intent → predicate logic assertions

Intent

model/product
Straightforward Observations

- **Problems**
  - formal proofs are long, tedious and are often hard; assertions are hard to get right; invariants are difficult to get right (need to be invariant, but also need to support overall proof strategy)
  - **Unsuccessful proof attempt** ⇒ ???
    - incorrect software? assertions? placement of assertion? inept prover? although failed proofs often indicate which of the above is likely to be true (especially to an astute prover)

- **Deeper Issues**
  - undecidability of predicate calculus ⇒ no way to be sure when you have a false theorem
  - there is no sure way to know when you should quit trying to prove a theorem (and change something)
  - proofs are generally much longer than the software being verified ⇒ errors in the proof are more likely than errors in the software being verified

Model Checking: Overview

- properties usually expressed in
  - in a propositional logic (e.g., temporal logic)
  - as a FSA
- system represented as a (possibly "abstracted") reachability graph
- reasoning engine
  - logic ⇒ propagates valid sub-formulas through the graph
  - FSA ⇒ compares FSAs via language inclusion; reachability; or bisimulation
Conservative Analysis

- If property is verified, property holds for all possible executions of the system
- If property is not verified:
  - an error found
  - OR
  - a spurious result
- System model abstracts information to be tractable
  - Conservative abstractions usually over-approximate behavior
  - If inconsistency relies upon over-approximations, then a spurious result
  - e.g. all counter example correspond to infeasible paths

Temporal logic

- augments the standard operators of propositional logic with "tense" operators
- "possible worlds semantics" ⇒ Kripke model
  - relativize the truth of a statement to temporal stages or states
  - a statement is not simply true, but true at a particular state
  - states are temporally ordered, with the type of temporal order determined by the choice of axioms.
- model of time
  - partially ordered time
  - linearly ordered time
  - linear temporal logic is typically extended by two additional operators, "until" and "since"
  - discrete time
  - branching (nondeterministic) time
    - foundation for one of the principal approaches to verifying concurrent systems = Computational Tree Logics.
Computation Tree Logics

- specification language
  - a propositional temporal logic.
- verification procedure
  - exhaustive search of the state space of the concurrent system to determine truth of specification.
- formulas constructed from path quantifiers and temporal operators:
  - path quantifier:
    - A "for every path"
    - E "there exists a path"
  - temporal operator:
    - Xp "p holds next time"
    - Fp "p holds sometime in the future"
    - Gp "p holds globally in the future"
    - pUq "p holds until q holds"

Architecture of FSV Systems

Property

System

Property Translator

Property Representation

System Model

Reasoning Engine

Property Verified

Counter Examples for Model
mutual exclusion protocol

- Example: processes can be null, trying to obtain the lock, or in a critical region (n1, t1, c1) or (n2, t2, c2)
  - TURN is a variable that indicates which process can obtain the lock (0, 1, 2)
  - Need a reachability graph that shows that states (i.e., the values) of the variables

```
Example: processes can be null, trying to obtain the lock, or in a critical region (n1, t1, c1) or (n2, t2, c2)
TURN is a variable that indicates which process can obtain the lock (0, 1, 2)
Need a reachability graph that shows that states (i.e., the values) of the variables
```

\[ \begin{align*}
\text{process1} &= n1, t1, c1 \\
\text{process2} &= n2, t2, c2 \\
\text{turn} &= 0, 1, 2
\end{align*} \]

Example: propagation

\[ \begin{align*}
AG(t1 \Rightarrow AF c1) \\
\text{t1} &\Rightarrow AF c1 \\
\text{t1} &\Rightarrow AF c1
\end{align*} \]

\[ \begin{align*}
\text{a \Rightarrow b means (b or \neg a)} \\
\text{\(( t1 \Rightarrow AF c1 )\) means ( AF c1 v \neg t1 )}
\end{align*} \]
Automata-Theoretic Model Checking

- Properties stated as an FSA

Intent

- Language containment
- Reachability analysis
- Bisimulation

Model/Product

FSA

Example

- Specification:
  - Of the possible observable events (a, b, c), c must happen at least once

\[(ba)^* (ac^* + bbc^*)\]

- Implementation

Accepted by?
Some observations

- Model Checking
  - worst case bound linear in size of the model
  - but the model is exponential
  - not clear if model checking or symbolic model checking is superior
  - depends on the problem
  - experimentally often very effective!
  - used selectively to verify hardware designs
  - trying to develop appropriate abstractions to make it applicable to software systems

Verification

- 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!
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

Types of Testing--what is tested

- Unit testing
  - exercise a single simple (procedure) component
- Integration testing
  - exercise a collection of inter-dependent components
  - focus on interfaces between components
- System testing
  - exercise a complete, stand-alone system
- Acceptance testing
  - customer’s evaluation of a system
  - usually a form of system testing
- Regression testing
  - exercise a changed system
  - Focus on modifications or their impact
Testing approaches

- “black box”

- “white box” or “glass box”

White & Black box testing processes
Testing Theory

- Specification-based C(s)
- Interface-based C(x1, x2, ..., xn, y1, y2, ..., yn)
- Program-based C(e)
- Combined C(s, e)

Criterion C for Test Adequacy

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>&quot;black box&quot;</th>
<th>&quot;white box&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification-based</td>
<td>C(s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interface-based</td>
<td>C(x1, x2, ..., xn, y1, y2, ..., yn)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program-based</td>
<td>C(e)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combined</td>
<td>C(s, e)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Types

- Structural
- Fault-based
- Error-based

If specification S defines a function F, such that P(F)Q, then C is reliable if T1, T2, ..., Tm; C(Ti, E); and D(E) ⊃ Ti.

\[
\begin{align*}
\forall T_i (\forall t \in T_i, E(t) = F(t)) & \lor \exists t \in T_i, E(t) \neq F(t) \\
(\exists t \in T_i, E(t) \neq F(t)) & \Rightarrow \exists t \in T_i, OK(T_i) \Rightarrow E = F
\end{align*}
\]
### Ideal Test Criterion

- test criterion C is *ideal* if for any executable component E and every test set $T_i \subseteq D(E)$ such that $C(T_i, E)$, $T_i$ is successful
  - of course we want $T_i \ll D(E)$
  - but in general, $T = D(P)$ is the only ideal test criterion

- In general, there is no ideal test criterion
  "Testing shows the presence, not the absence of bugs" – Dijkstra

- Dijkstra was arguing that verification was better than testing
- but, verification has similar problems
  - can’t prove an arbitrary program is correct
  - can’t solve the halting problem
  - can’t determine if the specification is complete

- need to use these techniques so that they compliment one another

### Black Box Testing

- **Functional/Interface Test Data Selection**
  - typical cases
  - boundary conditions/values
  - illegal conditions (if robust)
  - fault-revealing cases
    - based on intuition about what is likely to break the system
  - other special cases

- **stress testing**
  - large amounts of data
  - worse case operating conditions

- **combinations of events**
  - select those cases that appear to be more error-prone

- **common representations for selecting sequences of events**
  - decision tables
  - cause and effect graphs
  - usage scenarios
"White Box" Test Data Selection

- structural
  - coverage based
- fault-based
  - e.g., mutation testing, RELAY
- error-based
  - domain and computation based
  - use representations created by symbolic execution

CFG-Based Coverage

- Criteria
  - Statement Coverage
  - Path Coverage
  - Cyclomatic-number
  - Branch Coverage
  - Hidden Paths
  - Loop Guidelines
  - Boundary - Interior
- Selecting paths that satisfy the criteria
  - static selection
    - some of the associated paths may be infeasible
  - dynamic selection
    - monitors coverage and displays areas that have not been satisfactorily covered
“Simple” coverage measures

- **Statement Coverage**
  - Requires that each statement in a program be executed at least once
  - A set $P$ of paths in the CFG satisfies the statement coverage criterion if $\forall n_i \in \mathbb{N}, \exists p \in P$ such that $n_i$ is on path $p$

- **Path Coverage**
  - Requires that every path in the program be executed at least once
  - $P$ satisfies the path coverage criterion if $P$ contains all execution paths from the start node to the end node in the CFG
  - In most programs, path coverage is impossible

- **Multiple Condition Coverage**
  - $T$ is adequate if for every condition $C$ which consists of atomic predicates $(p_1, p_2, ..., p_n)$ and all possible combinations $(b_1, b_2, ..., b_n)$ of their values, there is at least one $t \in T$ such that the value of $p_i$ is equal to $b_i$ for all $i$

---

Branch & Loop Coverage

- **Branch Coverage**
  - Requires that each branch in a program (each edge in a control flow graph) be executed at least once
  - e.g., Each predicate must evaluate to each of its possible outcomes
  - Branch coverage is stronger than statement coverage

- **Loop Coverage**
  - Path 1, 2, 1, 2, 3 executes all branches (and all statements) but does not execute the loop well.
  - Better
    - fall through case
    - minimum number of iterations
    - minimum +1 number of iterations
    - maximum number of iterations
    - maximum -1 number of iterations

$X > 1$

$Y < 2$

hidden branches

1, 2, 1, 2, 1, 2, 3

$(1, 2)^{n-1}$, 3

$(1, 2)^n$, 3
Data Flow Path Selection

- **Definitions**
  - $d_n(x)$ denotes that variable $x$ is assigned a value at node $n$ (defined)
  - $u_m(y)$ denotes that variable $y$ is used (referenced at node $m$)
  - A definition clear path $p$ with respect to (wrt) $x$ is a subpath where $x$ is not defined at any of the nodes in $p$
  - A definition $d_n(x)$ reaches a use $u_m(x)$ if there is a subpath $(m \cdot p \cdot n)$ such that $p$ is definition clear wrt $x$

- **Rapps and Weyuker**
  - Definition-clear subpaths from definitions to uses

- **Ntafos**
  - Chains of alternating definitions and uses linked by definition-clear subpaths

- **Laski and Korel**
  - Combinations of definitions that reach uses at a node via a subpath

---

Rapps’ and Weyuker’s DF Criteria

- **All-Defs** - Some definition-clear subpath from each definition to some use reached by that definition

- **All-Uses** - Some definition-clear subpath from each definition to each use reached by that definition and each successor node of the use
### Rapps’ and Weyuker’s DF Criteria

- **All-C-Uses, Some-P-Uses**
  - C-use is a “computation use”
  - P-use is a “predicate use”
  - either All-C-Uses for \( dm(x) \) or at least one P-Use

- **All-P-Uses, Some-C-Uses**
  - either All-P-Uses for \( dm(x) \) or at least one C-Use

- **All-Du-Paths**
  - All definition-clear subpaths that are cycle-free or simple-cycles from each definition to each use reached by that definition and each successor node of the use.

### Examples

- **All-Defs Satisfactory Path:**
  - \( 1, 2, 4, 6 \)

- **All-Uses Satisfactory Paths:**
  - \( 1, 2, 4, 5, 6 \)
  - \( 1, 3, 4, 6 \)

- **All-Du-Paths Satisfactory Paths:**
  - \( 1, 2, 4, 5, 6 \)
  - \( 1, 3, 4, 5, 6 \)

- \( d_1(x) \) to any use
- \( d_1(x) \) to \( u_2(x) \)
- \( d_1(x) \) to \( u_3(x) \)
- \( d_1(x) \) to \( u_4(x) \)
- \( d_1(x) \) to \( u_5(x) \)
- both paths for \( d_1(x) \) to \( u_6(x) \)
Ntafos k-dr Data Flow Criteria

- Chains of alternating definitions and uses linked by definition-clear subpaths (k-dr interactions)
  - ith definition reaches ith use,
  - which defines (i+1)st definition
- Required K-tuples
  - Some subpath propagating each k-dr interaction
    + if last use is a predicate, both branches
    + if first definition or last use is in a loop, minimal and some larger number of loop iterations

3-DR interactions

\[ d_1(x), u_4(x), d_4(y), u_6(y) \]
\[ d_1(x), u_4(x), d_4(y), u_2(y) \]
\[ d_1(x), u_4(x), d_4(y), u_3(y) \]
\[ d_1(y), u_3(y), d_3(x), u_5(x) \]
\[ d_1(y), u_3(y), d_3(x), u_6(x) \]
\[ d_1(y), u_3(y), d_3(x), u_4(x) \]
\[ d_3(x), u_4(x), d_4(y), u_6(y) \]
\[ d_4(y), u_3(y), d_3(x), u_5(x) \]
\[ d_4(y), u_3(y), d_3(x), u_6(x) \]

Paths
  - 2-DR paths

©Rick Adrion 2005 (except where noted)
Laski’s and Korel’s Criteria

- Context Coverage - Some subpath along which each set of definitions reach uses at some node

- Ordered Context Coverage - Some subpath along which each ordering of each sequence of definitions reach uses at some node

Relationships among criteria

- All-Defs -- linear in assignment statements
- All-Uses -- quadratic in assignment statements
- All-DU-paths -- exponential in assignment statements, but empirically, all are linear in conditional statements
- Required 2-tuples -- quadratic in statements
- Reach -- linear in definitions that reach uses
- Context -- quadratic in definitions that reach uses
Fault-based Techniques

- Mutation Testing
  - introduce simple errors
  - apply test data to distinguish (kill)

- Fault propagation
  - apply test data to propagate fault to output

Mutation Testing

- Competent Programmer Hypothesis
  - programmers write programs that are reasonably close to the desired program
    - e.g., sort program is not written as a hash table
- Coupling Effect
  - detecting simple atomic faults will lead to the detection of more complex faults
- considers all simple (atomic) faults that could occur
  - introduces single faults one at a time to create "mutants" of original program
  - interactively (?) apply test data to complete (or partial) set of mutants
  - "test adequacy" is measured by "mutants killed"

operand mutations:
A := X + 1; ➞ A := X + 2 or ➞ A := X + Y

binary operator replacement:
A := X + 1; ➞ A := X - 1 or ➞ A := X * 1

statement replacement:
A := X + 1; ➞ continue or ➞ return
Mutation testing process

- Execute program P on test set T
  - save results R to serve as an oracle
  - P is considered the “correct” program
- Each fault results in a new program
  - Mutant programs = P1, ..., Pk
- Execute each mutant Pi on T and compare results Ri to R
  - If Ri ≠ R then mutant is killed
  - If Ri = R then either
    - Pi = P, thus it is an equivalent mutant
    - or the test cases do not reveal the error and need to find a new test case that does

Relay Model

```
transfer

:==

<op>

:==

fault

==

"observable" failure
```
Other Fault-Based techniques:

- mutating test data
- instead of mutating program, mutate input
- Bart Miller did an experiment where he demonstrated that arbitrary strings caused UNIX to consistently fail
  - wanted to understand why storms caused his connection to go down

Putting it all together

- unit testing
- integration & system testing
- regression testing
Unit testing

- test scaffolding
  - can be created for general or for specific tests
  - is composed of
    - one or more drivers
      - provide a prototype activation environment
      - drivers initialize non-local variables and parameters and call the unit
    - one or more stubs
      - provide a prototype of the units used by the program to be tested
    - one or more oracles
      - identify the tests that cause failures.

Unit vs. Integration vs. System Testing

- Integration testing
  - focuses on communication and interface problems
  - tests derived from module interfaces and detailed architecture specifications
  - some scaffolding is required

- System testing
  - focuses on the behavior of the system as a whole
  - tests are derived from requirements specifications
  - code is seen as a black box
  - support of scaffoldings not usually needed
    - exception is embedded code, where some simulation of the embedding environment may be required
Integration testing strategies

- big bang
  - [Diagram showing big bang strategy with Oracle and Outputs]

- top down
  - [Diagram showing top down strategy with Oracle and Outputs]

- bottom up
  - [Diagram showing bottom up strategy with Driver and Oracle]
Relation to design

- Big Bang
- Top Down
- Bottom Up
- Threads
- Critical Modules

Traditional
Incremental
Prototype (spiral)

Object Oriented Analysis & Design

- O-O Programs are Different
  - High Degree of Reuse - does this mean more, or less testing?
  - Unit Testing vs. Class Testing - what is the right "unit" in OO testing?
  - Classes appear early, so defects can be recognized early as well
- Unit Testing
  - What is a "Unit"?
    - Traditional: a "single operation"
  - Smallest testable unit = class
    - O-O: encapsulated data & operations
    - many operations
  - Inheritance
    - testing "in isolation" is impossible
    - operations must be tested every place they are used
Issues in O-O testing

- Primary Issues:
  - implications of encapsulation
  - implications of inheritance
  - implications of genericity
  - implications of polymorphism

- Changes concerns
  - State of instance variables
  - Sequences of methods calls
  - Must test a class and its specializations

Example

Base::describedSelf() is this code
if (val < 0) message("Less")
else if(val==0) message("Equal")
else message("More")

Tests:
- input, expected output
  -1 Less
  0 Equal
  1 More

Derived::describedSelf() is this code
if (val < 0) message("Less")
else if(val==0) message("Zero Equal")
else
{
  message("More")
  if(val==42) message("Jackpot")
}

Tests:
- input, expected output
  OK -1 Less
  Change 0 Equal Zero Equal
  OK 1 More
  Add 42 Jackpot
White-box vs. Black-box Testing

- The distance between object-oriented specification and implementation is typically small
- gap (and therefore usefulness) of the white-box/black-box distinction is decreasing
- But object-oriented specification describes functional behavior, while the implementation describes how that is achieved
- These techniques can be adapted to method testing, but are not sufficient for class testing
- Conventional flow-graph approaches
  - may be inconsistent the object-oriented paradigm
  - method-level control faults are not likely

Black-box O-O Testing

- Conventional black-box methods are useful for object-oriented systems
- error-guessing strategies
- verification of ADTs can be adapted to object-oriented systems
- Other approaches
  - utilize specifications integrated with the implementation as assertions
  - specification integrated with the implementation as dynamic assertions
  - C++ assertions or Eiffel pre/post-conditions offer similar self-checking
  - Utilize method (event) sequence information
    - usually don’t have history of method invocations
    - so can’t do this with assertions
Encapsulation

- not a source of errors but may be an obstacle to testing
- how to get at the concrete state of an object?
- use the abstraction
  - state is inspected via access methods
  - equivalence scenarios
  - comparing sequences of events
  - state is implicitly inspected by comparing related behavior
  - examine sequences of events
    - need to be able to define what are equivalent sequences and need to determine equal states
- use or provide hidden functions to examine the state
  - useful for debugging throughout the life of the system
  - but modified code, may alter the behavior
  - especially true for languages that do not support strong typing
- proof-of-correctness techniques
  - a proved method could be excused from testing to bootstrap testing of other methods
  - state reporting methods tend to be small and simple, they should be relatively easy to prove

Implications of Inheritance

- rule rather than the exception?
- inherited features usually require re-testing
  - because a new context of usage results when features are inherited
  - multiple inheritance increases the number of contexts to test
- specialization relationships
  - implementation specialization should correspond to problem domain specialization
  - reusability of superclass test cases depends on this
Which fns must be tested

Base class contains:
  inherited(int x)
  redefined() - returns a number in range 1 to 10 inclusive

Derived class contains:
  redefined() - returns a number in range 0 to 20 inclusive
  //inherited() is inherited

- derived::redefined has to be tested afresh
- does derived::inherited() have to be retested?

  inherited contains the code:
  if (x<0)
      x = x/redefined()
  return x

  have to test when x<0, could divide by 0

- derived::inherited() may not have to be completely tested
  - if code in inherited() doesn’t depend on redefined(), doesn’t call it nor call any code that indirectly calls it

Inheritance Testing

- flattening inheritance
  - each subclass is tested as if all inherited features were newly defined
  - tests used in the super-classes can be reused
  - many tests are redundant

- incremental testing
  - reduce tests only to new/modified features
  - determining what needs to be tested requires automated support
Polymorphism

- in procedural programming, procedure calls are statically bound
- each possible binding of a polymorphic component requires a separate set of test cases
- many server classes may need to be integrated before a client class can be tested
- may be hard to determine all such bindings
- complicates integration planning and testing

Testing under Inheritance

Q: What if implementation of `resize()` for each subclass calls inherited operation `move()`?

A: Shape cannot be completely tested unless we also test Circle, Square, & Ellipse!
Integration Testing

- OOS Not Hierarchical
  - Coupling is not via subroutine
  - "Top-down" and "Bottom-up" have little meaning
- Integrating one operation at a time is difficult
  - Indirect interactions among operations

- Thread-Based Testing
  - Integrate set of classes required to respond to one input or event
  - Integrate one thread at a time
  - Example: Event-Dispatching Thread vs. Event Handlers in Java
    - Implement & test all GUI events first
    - Add event handlers one at a time

- Use-Based Testing
  - Implement & test independent classes first
  - Then implement dependent classes (layer by layer, or cluster-based)
  - Simple driver classes or methods sometimes required to test lower layers

Test Case Design

- Focus
  - Designing sequences of operations to exercise the states of a class instance
- Challenges
  - Observability - Do we have methods that allow us to inspect the inner state of an object?
  - Inheritance - Can test cases for a superclass be used to test a subclass?
- Test Case Checklist
  - Identify unique tests & associate with a particular class
  - Describe purpose of the test
  - Develop list of testing steps:
    - Specified states to be tested
    - Operations (methods) to be tested
    - Exceptions that might occur
    - External conditions & changes thereto
    - Supplemental information (if needed)