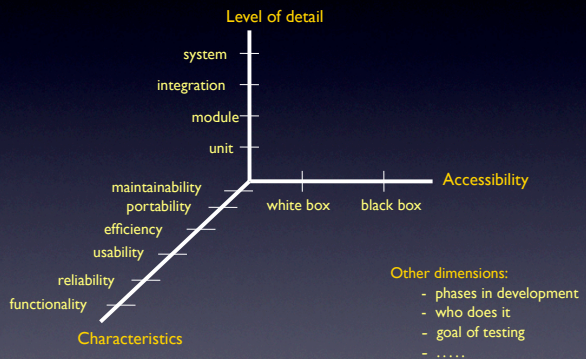

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a > c - b && a > b - c && b > a - c)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

More triangle tests...

1. is it fast enough ?
2. doesn't it use too much memory ?
3. is it learnable ?
4. is it usable for intended users ?
5. is it secure ?
6. does it run on different platforms ?
7. is it portable ?
8. is it easily modifiable ?
9. is the availability sufficient ?
10. is it reliable ?
11. does it comply with relevant laws ?
12. doesn't it do harm to other applications ?
13. . . . . .

functional testing, acceptance testing, duration testing, performance testing, interoperability testing, unit testing, black-box testing, white-box testing, grey-box testing, regression testing, reliability testing, usability testing, portability testing, security testing, compliance testing, recovery testing, integration testing, factory test, robustness testing, stress testing, conformance testing, developer testing, acceptance testing, production testing, module testing, system testing, alpha test, beta test, third-party testing, specification-based testing, ...

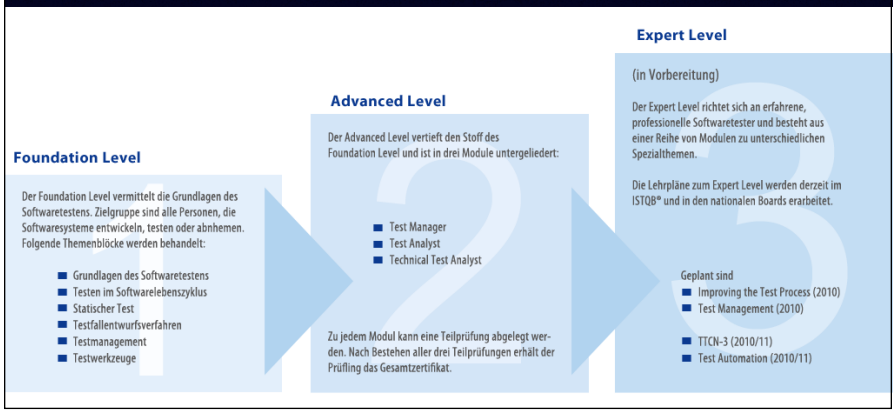
Sorts of Testing



Test Processes

- Requirements analysis
- Test planning
- Test development
- Test execution
- Test reporting
- Test result analysis
- Defect retesting
- Regression testing
- Test closure

Test Certification



Test Methods

- Exploratory testing
- Keyword testing
- Domain testing
- Scenario testing
- Capture & Replay
- GUI Testing
- Security testing
- Protocol testing
- Component-based testing
- Service testing
- Real-time testing
- Embedded systems testing
- Distributed testing
- Probabilistic testing
- Nondeterministic testing
- Metamorphic testing

Fuzzing

- One night (it was a dark and stormy night) in 1990, Bart Miller (U Wisc.) was logged in over dialup
- There was a lot of line noise due to the storm
- His shell and editors kept crashing
- This gave him an idea…

Fuzz Testing

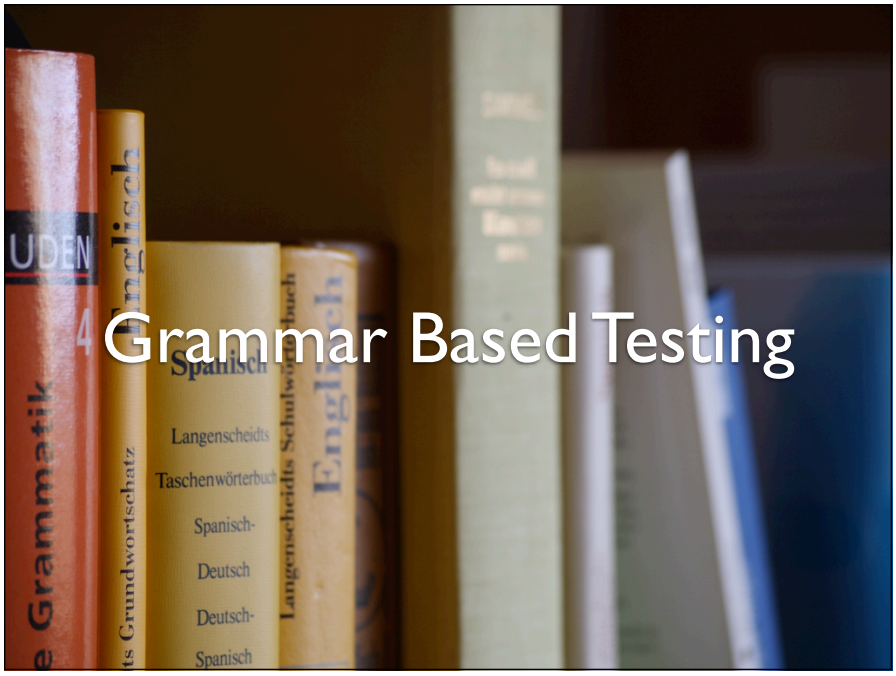
- Bart Miller et al., "An Empirical Study of the Reliability of UNIX Utilities"
- Idea: feed "fuzz" (streams of pure randomness, noise from /dev/urandom pretty much) to OS & utility code
- Watch it break!
- In 1990, could crash 25-33% of utilities
- Reports every few years since then
- Some of the bugs are the same ones in common security exploits (particularly buffer overruns)
- http://pages.cs.wisc.edu/~bart/

Fuzzers

- Tools that send malformed/random input to a program and hope to crash it or find a security hole
- Firefox is internally using random testing to find (security) problems
- Fuzzing is useful for finding bugs to protect programs ("white hat" work)
- But also for finding bugs to hack into systems ("black hat")!

Whitebox Fuzzing

- Essentially is dynamic symbolic execution
- E.g. done in large scale at Microsoft
- 100s machines running 24/7



Grammar Based Testing

Motivation

- Complex (textual) inputs
- Classical application: Testing compilers
- Web applications
- Interpreters
- Anything that uses XML
- Tree-like structures
- HTML injection vulnerabilities
- → Model the input

Context-free Grammars

- Finite set of terminals
- Finite set of nonterminals
- Finite set of rules
- Rule = Nonterminal → list of terminals and nonterminals
- Starting rule

```
expr → expr op expr
expr → ( expr )
expr → - expr
expr → id
op → +
op → -
op → *
op → /
op → ↑
```

Derivation

- Interpret production as rewriting rule
- Nonterminal on the left hand side is replaced by string on the right hand side
- E ⇒ -E ⇒ -(E) ⇒ -(id)
- Sequence of replacements = derivation

Grammar-based Testing

- Test generation = rewriting based on grammar
- Begin with start symbol
- Replace one nonterminal on the right with a rule with the nonterminal on the left
- Repeat until only terminals are left

History

- Hanford's Syntax Machine (1970)
- Earliest reported automated testing systems
- Inverse of a compiler
- Main application originally: testing compilers

Problems

- Context-free grammars - recursion
- Recursion - infinite number of possible inputs
- Which rule to replace with next?
- Which alternative to use for the replaced nonterminal?

Grammar Annotation

- Limit recursion depth
- Limit number of occurrences
- Limit parse tree depth
- Add output checks

Covering Grammars

- Terminal symbol coverage
  Each terminal must be used generate at least one test case
- Production coverage
  Each production must be used to generate at least one (section of) test case
- Boundary condition
  Annotate each recursive production with minimum and maximum number of application, then generate:
  - Minimum
  - Minimum + 1
  - Maximum - 1
  - Maximum

A Combinatorial Problem

- Testing VoIP software
- Caller, VoIP server, client
- CallerOS: Windows, Mac
- ServerOS: Linux, Sun, Windows
- CalleeOS: Windows, Mac

Same problem as grammar

```
Call ::= CallerOS ServerOS CalleeOS;
CallerOS ::= ‘Mac’;
CallerOS ::= ‘Win’;
ServerOS ::= ‘Lin’;
ServerOS ::= ‘Sun’;
ServerOS ::= ‘Win’;
CalleeOS ::= ‘Mac’;
CalleeOS ::= ‘Win’;
```

XML Schema

```
<xs:element name = "books">
   <xs:complexType>
   <xs:sequence>
      <xs:element name = "book" maxOccurs = "unbounded">
         <xs:complexType>
         <xs:sequence>
            <xs:element name = "ISBN" type = "isbnType" minOccurs = "0"/>
            <xs:element name = "author" type = "xs:string"/>
            <xs:element name = "title" type = "xs:string"/>
            <xs:element name = "publisher" type = "xs:string"/>
            <xs:element name = "price"  type = "priceType"/>
            <xs:element name = "year"   type = "yearType"/>
         </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
   </xs:complexType>
</xs:element>
```

```
<xs:simpleType name = "priceType">
   <xs:restriction base = "xs:decimal">
      <xs:fractionDigits value = "2" />
      <xs:maxInclusive value = "1000.00" />
   </xs:restriction>
</xs:simpleType>
```

Generating Tests

- Valid tests
  - Generate tests as XML messages by deriving strings from grammar
  - Take every production at least once
  - Take choices … "maxOccurs = "unbounded" means use 0, 1 and more than 1
- Invalid tests
  - Mutate the grammar in structured ways
  - Create XML messages that are "almost" valid

Mutants = Tests

Nonterminal Replacement
Every nonterminal symbol in a production is replaced by other nonterminal symbols.

Terminal and Nonterminal Duplication
Every terminal and nonterminal symbol in a production is duplicated.

Terminal Replacement
Every terminal symbol in a production is replaced by other terminal symbols.

Terminal and Nonterminal Deletion
Every terminal and nonterminal symbol in a production is deleted.

EXAMPLE

```
bank      ::=  action*
action    ::=  dep  |  deb
dep       ::=  “deposit” account amount
deb       ::=  “debit”  account amount
account   ::=  digit⁴
amount    ::=  “$” digit⁺ “.” digit²
digit     ::=  “0” | “1” | “2” | “3” | “4” | “5” | “6” |
               “7” | “8” | “9”
```


```
bank      ::=  action*
action    ::=  dep  |  deb
dep       ::=  “deposit” account amount
deb       ::=  “debit”  account amount
account   ::=  digit⁴
amount    ::=  “$” digit⁺ “.” digit²
digit     ::=  “0” | “1” | “2” | “3” | “4” | “5” | “6” |
               “7” | “8” | “9”
```


Nonterminal Replacement

```
dep ::= “deposit” account amount
dep ::= “deposit” amount amount
dep ::= “deposit” account digit
```

deposit $1500.00 $3789.88
deposit 4400 5


```
bank      ::=  action*
action    ::=  dep  |  deb
dep       ::=  “deposit” account amount
deb       ::=  “debit”  account amount
account   ::=  digit⁴
amount    ::=  “$” digit⁺ “.” digit²
digit     ::=  “0” | “1” | “2” | “3” | “4” | “5” | “6” |
               “7” | “8” | “9”
```


Terminal Replacement

```
amount ::= “$” digit+ “.” digit²
amount ::= “.” digit+ “.” digit²
amount ::= “$” digit+ “$” digit²
amount ::= “$” digit+ “1” digit²
```

deposit 4400 .1500.00
deposit 4400 $1500$00
deposit 4400 $1500100

```
bank      ::=  action*
action    ::=  dep  |  deb
dep       ::=  “deposit” account amount
deb       ::=  “debit”  account amount
account   ::=  digit⁴
amount    ::=  “$” digit⁺ “.” digit²
digit     ::=  “0” | “1” | “2” | “3” | “4” | “5” | “6” |
               “7” | “8” | “9”
```

Terminal and Nonterminal Duplication

```
dep ::= “deposit” account amount
dep ::= “deposit” “deposit” account amount
dep ::= “deposit” account account amount
dep ::= “deposit” account amount amount
```

deposit deposit 4400 $1500.00
deposit 4400 4400 $1500.00
deposit 4400 $1500.00 $1500.00


```
bank      ::=  action*
action    ::=  dep  |  deb
dep       ::=  “deposit” account amount
deb       ::=  “debit”  account amount
account   ::=  digit⁴
amount    ::=  “$” digit⁺ “.” digit²
digit     ::=  “0” | “1” | “2” | “3” | “4” | “5” | “6” |
               “7” | “8” | “9”
```


Terminal and Nonterminal Deletion

```
dep ::= “deposit” account amount
dep ::= account amount
dep ::= “deposit” amount
dep ::= “deposit” account
```

4400 $1500.00
deposit $1500.00
deposit 4400

Generating Mutants

- We have more experience with program-based mutation than input grammar based mutation
  - Operators are less “definitive”
- Applying mutation operators
  - Mutate grammar, then derive strings
  - Derive strings, mutate a derivation “in-process”
  - Some mutants give strings in the original grammar (equivalent)
  - These strings can easily be recognized to be equivalent

Test Generation with Model Checkers

TODO: Examples - Intel etc?

- Model checking

  Exhaustive state space exploration
- Input
  Model + property
- Output
  Proof of correctness or counterexample
- Successfully applied in (HW) industry

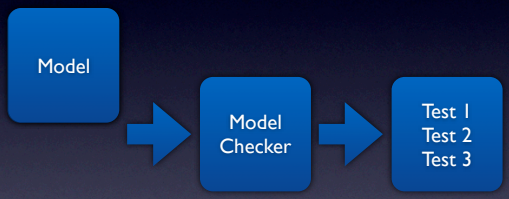
Temporal Logic

- Boolean logics + temporal operators
- For our purposes:
  - Temporal operators: Globally (G), Next (X)
  - Path quantifier: Always (A)
- AG(condition)
  Condition always holds along all execution paths
- AG(condition1 -> AX condition2)
  Condition 1 always implies condition 2 on all successive states along all paths

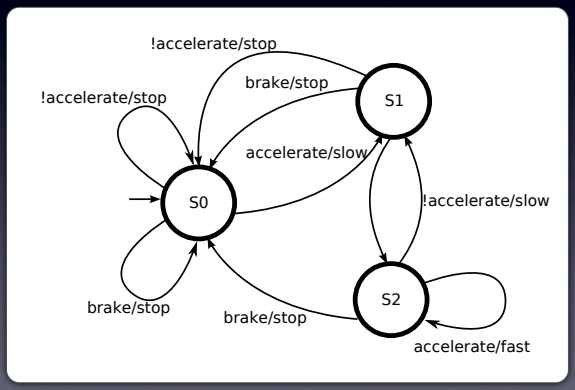
Test Specifications in Temporal Logics

- Counterexample shows violation of property
- Counterexample = execution sequence
- Property "My test goal is not reachable!"
  "Trap property"
- Counterexample = test case

Testing with Model Checkers



Example



Example: SMV

```
MODULE main
VAR
  accelerate: boolean;
  brake: boolean;
  velocity: { stop, slow,fast };

ASSIGN
  init(velocity) := stop;
  next(velocity) := case
     accelerate  & !brake & velocity = stop : slow;
     accelerate  & !brake & velocity = slow : fast;
     !accelerate & !brake & velocity = fast : slow;
     !accelerate & !brake & velocity = slow : stop;
     brake: stop;
     TRUE : velocity;
  esac;
```

Example

- Coverage criteria
  AG(!condition)
- Mutation testing
  AG(!(condition xor mutant))
- Combinatorial testing
  AG !(var1 = val1 & var2 = val2)

Web Testing

Based on slides by Jeff Offutt & Paul Ammann

Issues in Testing Web Software

- A *web application* is a program that is deployed on the web
  - Usually uses HTML as the user interface
  - Web-deployment means they are available worldwide
  - They accept requests through HTTP and return responses
  - HTTP is stateless – each request/response pair is independent
- Web applications are usually very competitive
- A *web service* is a web-deployed program that accepts XML messages wrapped in SOAP
  - Usually no UI with humans
  - Service must be published so other services and applications can discover them

Web Software

- Composed of independent, loosely coupled software components
  - All communication is through messages
  - Web application messages always go through clients
  - The only shared memory is through the session object – which is very restricted
  - The definition of state is quite different
- Inherently concurrent and often distributed
- Most components are relatively small
- Uses numerous new technologies, often mixed together

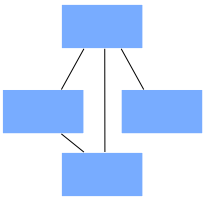
Problem Parameters

- HTTP is a stateless protocol
  - Each request is independent of previous request
- Servers have little information about where a request comes from
- Web site software is extremely loosely coupled
  - Coupled through the Internet – separated by space
  - Coupled to diverse hardware devices
  - Written in diverse software languages

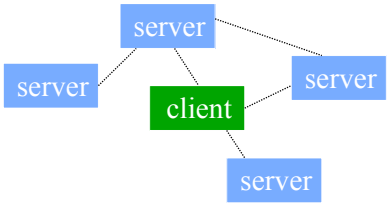
Differences in Testing Web Software

- Traditional graphs do not apply
  - Control flow graph
  - Call graph
- State behavior is hard to model and describe
- All inputs go through the HTML UI – low controllability
- Hard to get access to server-side state (memory, files, database) – low observability
- Not clear what logic predicates can be effectively used
- No model for mutation operators on web software

Problem 1: Loosely Coupled



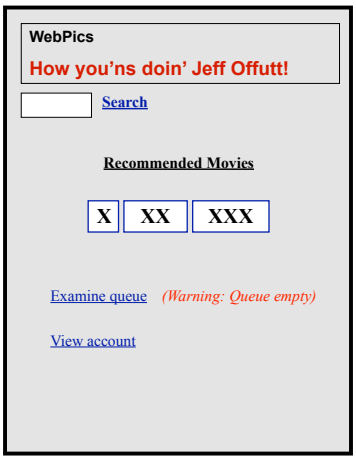
Traditional software
Connected by calls and message passing
High and moderate coupling

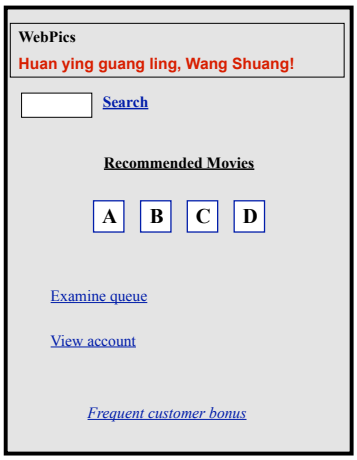


Web-based software
Connected with HTTP and XML
Loose, extremely loose, distributed coupling

How can we ensure the reliability of this type of software?

Problem 2: Dynamic Flow of Control





How can we ensure the reliability of this type of system?

Dynamic Execution of Web Apps

- Parts of the program are generated dynamically
- Dynamic web pages are created when users make requests
- Different users will see different programs !
- The potential control (the traditional control flow graph) cannot be known ahead of time

Problem 3: User Control Flow

- Users can make unexpected changes to the flow of control

  Back buttons, refreshing, caching, forward button, URL hacking
- State is stored in the server and in the HTML in the client's browser
- ***Operational transitions*** : Transitions NOT based on an HTML link or program statement: back, forward, URL rewriting, refresh

These transitions can cause unanticipated changes to the state of the web application

How can we ensure the reliability of this type of system?

Problem 4: Dynamic Integration

- Software modules can dynamically integrate with others if they use the same data structures
- EJBs can be inserted into web applications, which can immediately start using them
- Web services find and bind to other web services dynamically

New Essential Problems of Web Apps

- Web site applications feature distributed integration and are extremely loosely coupled

  Internet and diverse hardware / software

- HTML forms are created dynamically by web applications

  UI created on demand and can vary by user and time

- Users can change the flow of control arbitrarily

  back button, forward button, URL rewriting, refresh

- Dynamic integration of new software components

  new components can be added during execution

Testing Static Hyper Text Web Sites

- This is not program testing, but checking that all the HTML connections are valid
- The main issue to test for is dead links
- We should also evaluate
  - Load testing
  - Performance evaluation
  - Access control issues
- The usual model is that of a graph
  - Nodes are web pages
  - Edges are HTML links

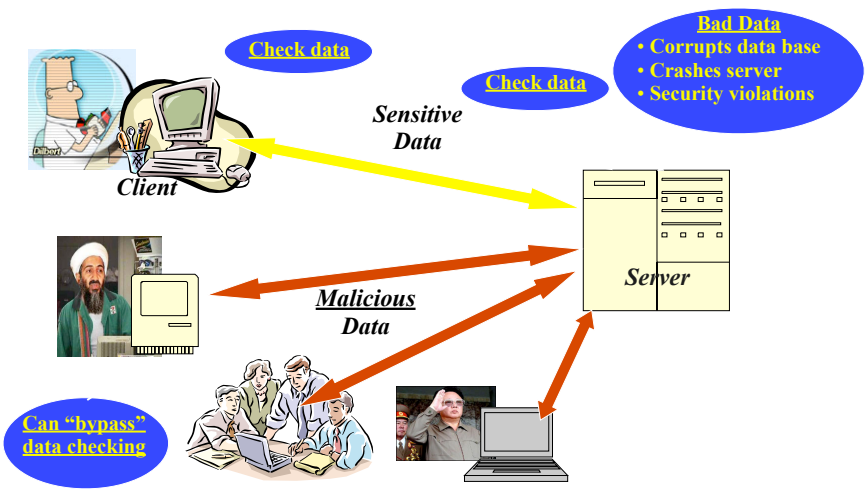
Testing Dynamic Hyper Text Web Sites

- The user interface is on the client
- Some software is on the client (scripts such as Javascript)
- Most software is on the server
- Client-side testing does not access source or state on the server
- Server-side testing can use the source or the server state

Client-Side (Black-Box) Testing

- The UI and the software are on separate computers
- The inputs to web software are defined by the HTML form elements
  - Text boxes, buttons, dropdown lists, links, etc
- Techniques for generating values
  - Supplied by the tester
  - Generated randomly
  - User session data – data collected from previous users of the software
- Choosing values
  - Bypass testing – values that violate constraints on the inputs, as defined by client-side information
- The problem of finding all screens in a web application is undecidable

Web Application Input Validation

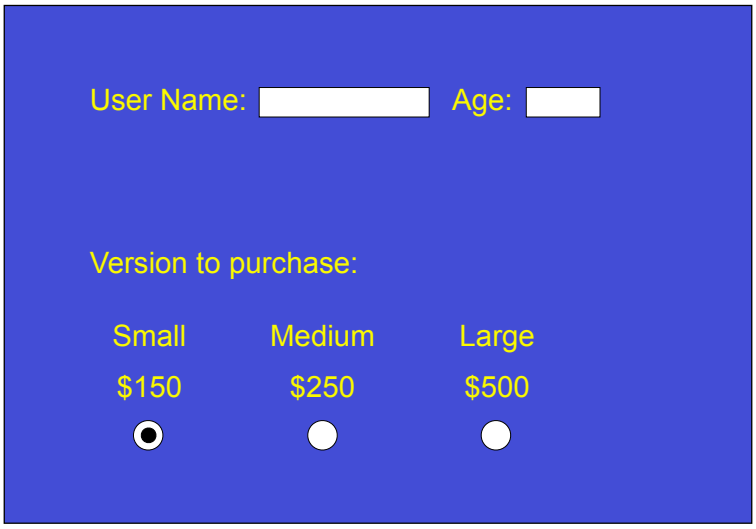


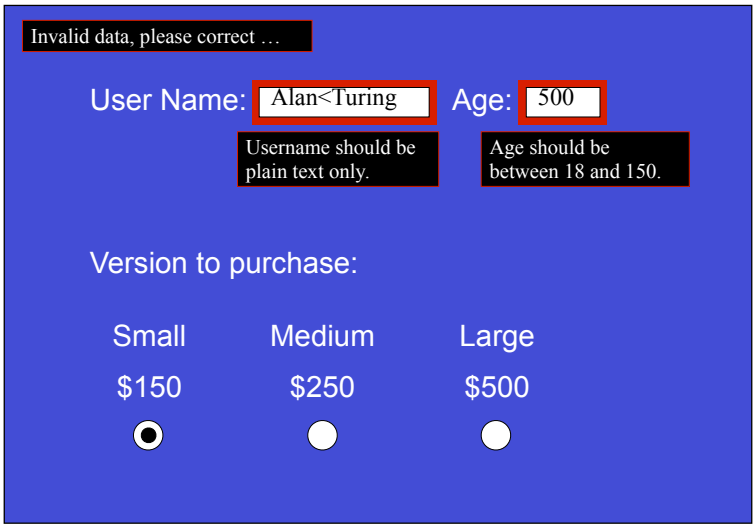
Bypass Testing

"***bypass***" client-side constraint enforcement

Bypass testing constructs tests to intentionally violate constraints :

- Eases test automation
- Validates input validation
- Checks robustness
- Evaluates security





Abbreviated HTML

```
<FORM >
  <INPUT Type="text" Name="username" Size=20>
  <INPUT Type="text" Name="age" Size=3 Maxlength=3>
  <P> Version to purchase:
     …
  <INPUT Type="radio" Name="version" Value="150" Checked>
  <INPUT Type="radio" Name="version" Value="250">
  <INPUT Type="radio" Name="version" Value="500">
  <INPUT Type="submit" onClick="return checkInfo(this.form)">
  <INPUT Type="hidden"  isLoggedIn="no">
</FORM>
```

Bypass Behavior

Extremely loose coupling …

combined with the stateless protocol …

allows users to easily bypass client-side checking :

Saved & Modified HTML

```
<FORM >
  <INPUT Type="text" Name="username" Size=20>
  <INPUT Type="text" Name="age" Size=3 Maxlength=3>
  <P> Version to purchase:
    …
  <INPUT Type="radio" Name
  <INPUT Type="radio" Name="version" Value="250" >
  <INPUT Type="radio" Name="version" Value="25" Checked>
  <INPUT Type="submit" onClick="return checkInfo (this.form)">
   <INPUT Type="hidden"  isLoggedIn= yes >
</FORM>
```

Allows an input with arbitrary age, no checking, cost=$25 …

'<' can crash an XML parser

Text fields can have SQL statements

SQL Injection

User Name: turing ' OR '1'='1 **Password:** enigma ' OR '1'='1

Original SQL:

SELECT username FROM adminuser WHERE username='turing' AND password ='enigma'

"injected" SQL:

SELECT username FROM adminuser WHERE username='turing' OR '1' = '1' AND password ='enigma' OR '1' = '1'

Bypass Testing

This example illustrates how users can "bypass" client-side constraint enforcement

Bypass testing constructs tests to intentionally violate constraints

- Eases test automation
- Checks robustness
- Evaluates security

Preliminary results

- Rules for constructing tests
- Successfully found errors in numerous Web apps

Example Client-Side Constraint Rules

Violate size restrictions on strings

Introduce values not included in static choices

- Radio boxes
- Select (drop-down) lists

Violate hard-coded values

Use values that JavaScripts flag as errors

Change "transfer mode" (get, post, …)

Change destination URLs

Server-Side (White-Box) Testing

If we have access to the source on the server, we can try to model the web software

Many testing criteria on non-web software rely on a static control flow graph

- Edge testing, data flow, logic coverage, …
- Also slicing, change impact analysis, …

The standard control flow graph cannot be computed for web applications !

- But all the pieces of the web pages and the programs are contained in the software presentation layer …

Atomic Sections

A section of HTML with the property that if any part of the section is sent to a client, the entire section is

May include JavaScript

– All or nothing property

An HTML file is an atomic section

- ***Content variable*** : A program variable that provides data to an atomic section

Atomic sections may be empty

Component Expressions

Atomic sections are combined to model dynamically generated web pages

Four ways to combine:

Sequence : $p1 \bullet p2$

Selection : $(p1 \mid p2)$

Iteration : $p1^*$

Aggregation : $p1\ \{p2\}$

– $p2$ is included inside of $p1$

The previous example produces:

$p \rightarrow p1 \bullet (p2 \bullet (p3 \mid p4)^* \mid p5) \bullet p6$

Composite sections can be produced automatically

Modeling Component Transitions

Five types of transitions

1. Simple Link Transition: An HTML link (<A> tag)
2. Form Link Transition: Form submission link
3. Component Expression Transition: Execution of a software component causes a component expression to be sent to the client
4. Operational Transition: A transition out of the software's control
   - Back button, Forward button, Refresh button, User edits the URL, Browser reloads from cache
5. Redirect Transition: Server side transition, invisible to user

Modeling Web Applications

Restricted to the presentation layer only

Two levels of abstraction

1. Component Interaction Model (CIM)
   - Models individual components
   - Combines atomic sections
   - Intra-component
2. Application Transition Graph (ATG)
   - Each node is one CIM
   - Edges are transitions among CIMs
   - Inter-component

Component Interaction Model : gradeServlet

```
ID             = request.getParameter ("Id");
passWord       = request.getParameter ("Password");
retry          = request.getParameter ("Retry");
PrintWriter out = response.getWriter();
P1 = out.println (“<HTML> <HEAD><TITLE>" +  title   + "</TITLE></HEAD><BODY>)"
     if ((Validate (ID, passWord)) {
P2 =     out.println (“ <B> Grade Report </B>");
         for (int I=0; I < numberOfCourse; I++)
P3 =         out.println(“<p><b>" +  courseName (I)   + "</b>“ + courseGrade (I) + “</p>”);
     } else if (retry < 3) {
P4 =     retry++;
         out.println ("Wrong ID or wrong password");
         out.println ("<FORM Method=\“get\" Action=\"gradeServlet\">);
         out.println ("<INPUT Type=\“text\" Name=\"Id\" Size=10>");
         out.println ("<INPUT Type=\“password\" Name=\"Password\" Width=20>");
         out.println ("<INPUT Type=\“hidden\" Name=\"Retry\" Value=" + (retry) + ">");
         out.println ("<INPUT Type=\“submit\" Name=\"Submit\" Value=\“submit\">");
         out.println ("<A Href=\"sendMail\">Send mail to the professor</A>");
     } else if (retry >= 3) {
P5 =     out.println (“<p>Wrong ID or password, retry limit reached. Good bye.")   }
P6 = out.println(“</BODY></HTML>");
```

CIM for gradeServlet

S = login.html

$A = \{p_1, p_2, p_3, p_4, p_5, p_6\}$

CE = gradeServlet $= p_1 \bullet ((p_2 \bullet p_3^{*}) \mid p_4 \mid p_5) \bullet p_6$

$T = \{$login.html $\longrightarrow\!\!>$ gradeServlet [get, (Id, Password, Retry)],

gradeServlet.p_4 $\longrightarrow$ sendMail [get, ()],

gradeServlet.p_4 $\longrightarrow\!\!>$ gradeServlet [get, (Retry)] $\}$

Application Transition Graph

Γ : Finite set of web components

Θ: Set of transitions among web software components

Includes type of HTTP request and data

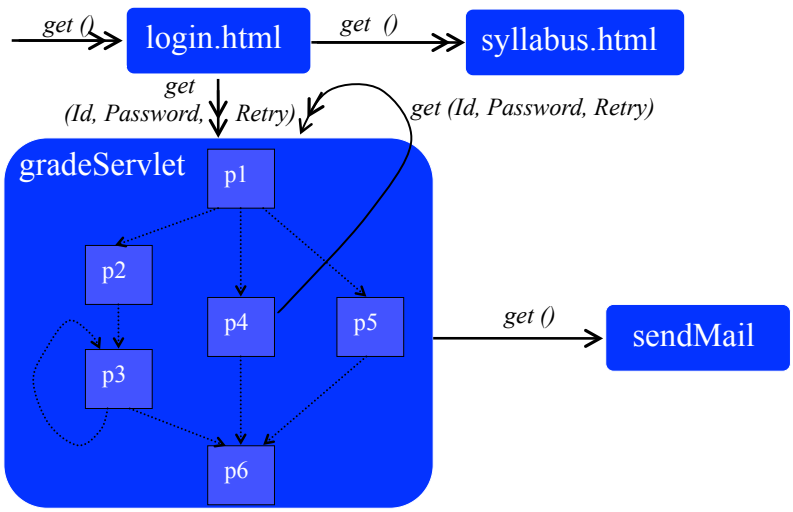
Σ: Set of variables that define the web application state

α: Set of start pages

ATG for gradeServlet

- Γ = { login.html, gradeServlet, sendMail, syllabus.html }
- Θ = {login.html ⟶ syllabus.html [get, ()],
  login.html ⟶⟶ gradeServlet [get, (Id, Password, Retry)],
  gradeServlet.p_4 ⟶ sendMail [get, ()],
  gradeServlet.p_4 ⟶⟶ gradeServlet [get, (Retry)] }
- Σ = { Id, Password, Retry }
- α = { login.html }

ATG for gradeServlet



Web Services

A ***Web Service*** is a program that offers services over the Internet to other software programs

- Internet-based

  Uses SOAP and XML

- Peer-to-peer communication

Web service components can integrate dynamically, by finding other services during execution

Web services transmit data that are formatted in XML

Difficulties of Testing Web Services

Web services are always distributed

Most "peer-to-peer" communication is between services published by different organizations

- ***Trust*** is a major issue holding back the adoption of web services !

- Design and implementation are almost never available
- Structured messages are transmitted

Most testing research so far has focused on messages

- Syntax-based test criteria have been proposed for Web services

Conclusions

The Web provides a new way to deploy software

Web applications:

offer many advantages

use many new technologies

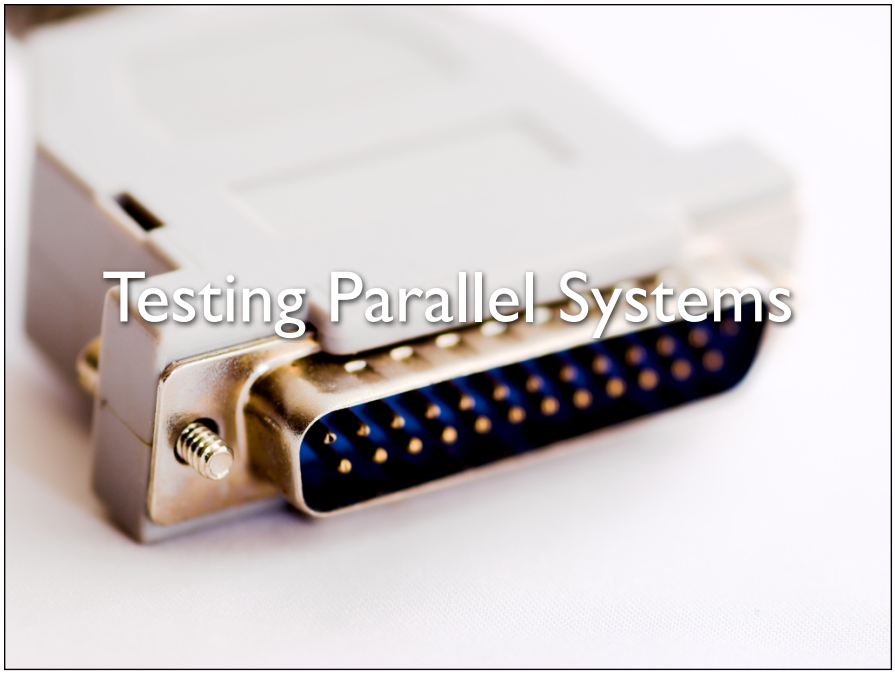
introduce fascinating new problems

- Web software engineering is just starting

Two very useful techniques:

- Atomic sections : A fundamental model
- Bypass testing : Easy to automate – no source needed

This is a very active research area



Concurrency

- Concurrent Programming is HARD
- Concurrent executions are highly nondeterminisitic
- Rare thread interleavings result in Heisenbugs
- Difficult to find, reproduce, and debug
- Observing the bug can "fix" it
- Likelihood of interleavings changes, say, when you add printfs
- A huge productivity problem
- Developers and testers can spend weeks chasing a single Heisenbug

Sources of Nondeterminism

- Interleaving nondeterminism
- Threads can race to access shared variables or monitors
- OS can preempt threads at arbitrary points
- Timing nondeterminism
- Timers can fire in different orders
- Sleeping threads wake up at an arbitrary time in the future
- Asynchronous calls to the file system complete at an arbitrary time in the future

Sources of Nondeterminism

- Hardware relaxations
- The processor can reorder memory instructions
- Can potentially introduce new behavior in a concurrent program
- Compiler relaxations
- Compiler can reorder memory instructions
- Can potentially introduce new behavior in a concurrent program (with data races)

Race Condition

- Occurs when two or more threads of execution in a multi-threaded program try to access the same shared data and at least one of the accesses is a write.
- Harmful race conditions introduce unpredictability and are often hard to detect
- The consequences of a race condition might only become visible at a much later time or in a totally different part of the program
- They are also incredibly hard to reproduce
- Races are avoided by using synchronization techniques to correctly sequence operations between threads

Deadlock

- Two or more threads wait on each other
- Forming a cycle and preventing all of them from making any forward progress
- Can be introduced by the programmer while trying to avoid race conditions
- For example, incorrect use of synchronization primitives such as acquiring locks in an incorrect order can result in two or more threads waiting for each other
- Deadlocks can also occur in cases that don't use locking constructs; any kind of circular wait can result in a deadlock.

Starvation

- Starvation is an indefinite delay or permanent blocking of one or more runnable threads in a multithreaded application
- Threads that are not being scheduled to run even though they are not blocking or waiting on anything else are said to be starving
- Starvation is typically the result of scheduling rules and policies
- For example, if one schedules a high-priority, non-blocking, continuously executing thread along with a low-priority thread, then, on a single core CPU, the lower thread will never get a chance to run
- To help avoid such conditions, the Windows scheduler intervenes frequently to reduce starvation by occasionally boosting priorities of starving threads.

Livelocks

- Livelocks occur when threads are scheduled but are not making forward progress because they are continuously reacting to each other's state changes
- The best way to describe this is if two people were to approach each other in a narrow hallway and they step aside for one another, each time blocking the other's path
- Such side stepping prevents forward progress and results in a live lock
- High CPU utilization with no sign of real work being done is a classic warning sign of a livelock
- Livelocks are incredibly difficult to detect and diagnose

Lockset Algorithm

1. For each shared memory variable v, maintain a set C(v) of locks. Initially C(v) is the list of all locks.
2. Each thread also maintains two sets of locks: locks(t) indicating all the locks held and writeLocks(t) indicating all the write locks held.
3. On each read of v by thread t:
    1. Set C(v) = C(v) intersection locks(t).
    2. If C(v) == NULL set, then raise an error.
4. On each write of v by thread t:
    1. Set C(v) = C(v) intersection writeLocks(t).
    2. If C(v) ==NULL set, then raise an error.

Lockset Algorithm

- As the application progresses, the C(v) of each variable starts shrinking
- An error is raised if the C(v) ever becomes null
  E.g. if the intersection of the threads' locksets are NULL at the time of accessing the shared memory
- Not all races reported by a lockset algorithm are real races
  One can write race-free code without using locks either by applying clever programming tricks or using other synchronization primitives such as signaling
- Annotations and certain suppressions can help alleviate this problem

Testing Concurrent Systems

- State-less model checking
  E.g., CHESS exploration
- Tester controls the scheduler
- Re-run same test with different scheduling
- Exhaustive exploration possible
  under certain assumptions

Design by Contract

- A *contract* of a method describes
  - what the method *requires* (precondition)
  - what the method *provides* (postcondition)
- The contract binds clients (method callers) and suppliers (method implementors)

A Stack

```
class STACK[G]      -- A stack of G's
    count: INTEGER
    empty: BOOLEAN -- true if empty
    full: BOOLEAN  -- true if full
    put(x: G)      -- add x to stack
        require    -- precondition
            not full
        do …       -- implementation
        ensure     -- postcondition
            not empty
            item = x
            count = old count + 1
        end
end
```

Checking Contracts

- Contracts are *checked at runtime*
- Failing contracts indicate internal errors (and therefore raise exceptions)
- Contracts guarantee *correctness* – *if* the program terminates
- Can also be used for program proofs (as Z)

Rights and Obligations

If you promise to call *m* with *pre* satisfied, then I, in return, promise to deliver a final state in which *post* is satisfied.

Design by Contract

Method	Obligations	Benefits
Client	Satisfy precondition	Obtain postcondition
Supplier	Satisfy postcondition	Rely on precondition

Design by Contract

put(x)	Obligations	Benefits
Client	Only call put(x) on a non-full stack	• Stack is updated • x is on top • Count is increased
Supplier	• Update stack • Put x on top • Increase count	Need not check whether stack is full

Contract Violations

- A violation in the *precondition* indicates a defect in the *client*
- A violation in the *postcondition* indicates a defect in the *supplier*
- Useful for locating defects (and for putting the blame on someone)

Testing Contracts

- When testing a certain method:
  - We try to satisfy its precondition (so that we can execute it)
  - We try to violate its postcondition => bugs

```
class ARRAYED_LIST [G] ...
    put (v: like item) is
            -- Replace current item by `v'.
            -- (Synonym for `replace')
        require
            extendible: extendible
        do
            ...
        ensure
            item_inserted: is_inserted (v)
            same_count: count = old count
        end
```

precondition

body

postcondition