Lecture 7

Advanced Topics in Testing:

(i) Mutation Testing

(ii) Learning-based Testing

Mutation Testing

- Mutation testing concerns:

> evaluating test suites for their inherent quality,
> i.e. ability to reveal errors.

- Need an objective method to measure quality
- Need to define "what is an error"
- Basic idea is to inject defined errors into the SUT and evaluate whether a given test suite finds them.
- "*Killing a mutation*"

Basic Measurement Idea

- We can statistically estimate (say) fish in a lake by releasing a number of marked fish, and then counting the number of marked fish in a subsequent small catch.
- Example: release 20 fish
- Catch 40 fish, 4 marked.
- Then 1 in 10 is marked, so estimate 200 fish
- Pursue the same idea with marked SW bugs?

Mutations and Mutants

- The “marked fish” are injected errors, termed mutations
- The mutated code is termed a mutant
- Example: replace < by > in a one Boolean expression
- if (x < 0) then … becomes if (x > 0) then …
- If test suite finds mutation we say this particular mutant is killed
- Make large set of mutants – typically using a checklist of known mutations - mutation tool

Mutation score

- Idea is that if we can kill a mutant we can identify a real bug too
- Mutants which are semantically equivalent to the original code are called equivalents
- Write $Q \equiv P$ if Q and P are equivalents
- Clearly cannot kill equivalents
- Mutation score % =

$$\frac{\text{Number of killed mutants}}{\text{total number of non-equivalent mutants}} * 100$$

Why should it work?

- Two assumptions are used in this field

Competent programmer hypothesis

i.e. "The program is mostly correct"

and the

Coupling Effect

Semantic Neighbourhoods

- Let Φ be the set of all programs semantically close to P (defined in various ways)
- Φ is neighbourhood of P
- Let T be a test suite, $f:D \rightarrow D$ be a functional spec of P
- Traditionally assume something like:

$$\forall t \in T\ P.t = f(t) \rightarrow \forall x \in D\ P.x = f(x)$$

- i.e. T is a reliable test suite
- Requires exhaustive testing

Competent Programmer Hypothesis

incompetent

- P is pathological iff $P \notin \Phi$
- Assume programmers have some competence

> Mutation testing assumption
>
> Either P is pathological or else
>
> $\forall t \in T\ P.t = f(t) \rightarrow \forall x \in D\ P.x = f(x)$

- Can now focus on building a test suite T that would distinguish P from all other programs in Φ

Coupling Effect

- The competent programmer hypothesis limits the problem from infinite to finite.
- But remaining problem is still too large

Coupling effect says that there is a small subset $\mu \subseteq \Phi$ such that:

We only need to distinguish P from all programs in μ by tests in T

Problems

- Can we be sure the coupling effect holds? Do simple syntactic changes define a set?
- Can we detect and count equivalents? If we can't kill a mutant Q is $Q \equiv P$ or is Q just hard to kill?
- How large is μ? May still be too large to be practical?

Equivalent Mutants

- Offut and Pan [1997] estimated 9% of all mutants equivalent
- Bybro [2003] concurs with 8%
- Automatic detection algorithms (basically static analysers) detect about 50% of these
- Use theorem proving (verification) techniques

Coupling Effect

- For Q an incorrect version of P
- Semantic error size = $\Pr\{Q.x \neq P.x\}$
- If for every semantically large fault there is an overlap with at least one small syntactic fault then the coupling effect holds.
- i.e. big errors are always coupled to small errors.
- Selective mutation based on a small set of semantically small errors - “Hard to kill”

Early Research: 22 Standard (Fortran) mutation operators

AAR Array reference for array reference replacement
ABS Absolute value insertion
ACR Array reference for constant replacement
AOR Arithmetic operator replacement
ASR Array reference for scalar replacement
CAR Constant for array reference replacement
CNR Comparable array name replacement
CRP Constants replacement
CSR Constant for Scalar variable replacement
DER Do statement End replacement
DSA Data statement alterations
GLR Goto label replacement
LCR Logical connector replacement
ROR Relational operator replacement
RSR Return statement replacement
SAN Statement analysis
SAR Scalar for array replacement
SCR Scalar for constant replacement
SDL Statement deletion
SRC Source constant replacement
SVR Scalar variable replacement
UOI Unary operator insertion

Recent Research: Java Mutation Operators

- First letter is category
- A = access control
- E = common programming mistakes
- I = inheritance
- J = Java-specific features
- O = method overloading
- P = polymorphism

AMC Access modifier change

EAM Accessor method change

EMM Modifier method change

EOA Reference assignment and content assignment replacement

EOC Reference comparison and content comparison replacement

IHD Hiding variable deletion

IHI Hiding variable insertion

IOD Overriding method deletion

IOP Overriding method calling position change

IOR Overridden method rename

IPC Explicit call of parent's constructor deletion

ISK super keyword deletion

JDC Java supported default constructor create

JID Member variable initialisation deletion

JSC static modifier change

JTD this keyword deletion

OAO Argument order change

OAN Argument number change

OMD Overloaded method deletion

OMR Overloaded method contents change

PMD Instance variable deletion with parent class type

PNC new method call with child class type

PPD Parameter variable declaration with child class type

PRV Reference asssignment with other compatible type

Practical Example

- Triangle program
- Myers “complete” test suite (13 test cases)
- Bybro [2003] Java mutation tool and code
- 88% mutation score, 96% statement coverage

Status of Mutation Testing

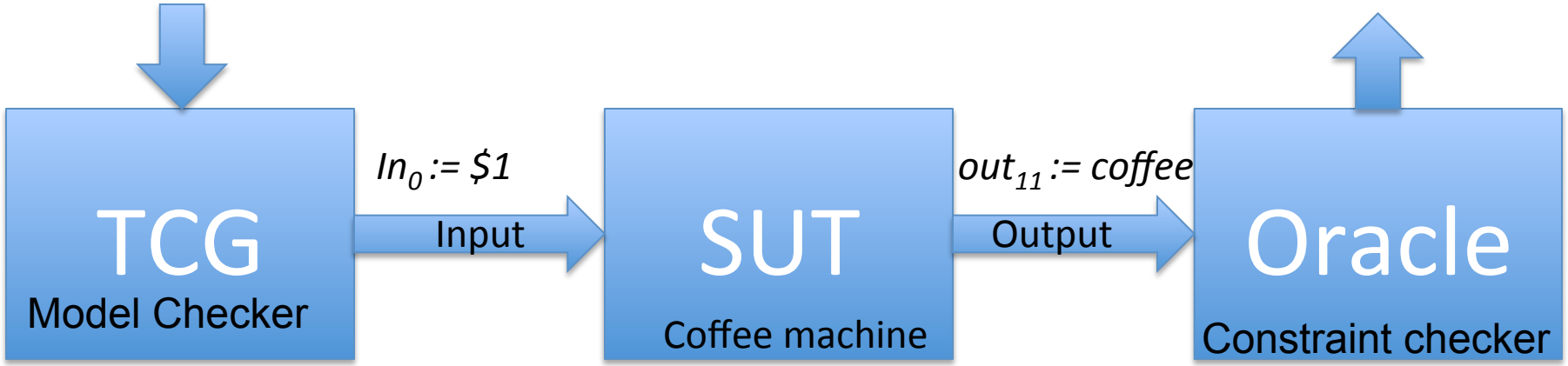
- Various strategies: weak mutation, interface mutation, specification-based mutation
- Our version is called strong mutation
- Many mutation tools available on the internet
- Cost of generating mutants and detecting equivalents has come down
- Not yet widely used in industry
- Still considered “academic”, not understood?

Learning-based Testing

- Learning-based testing is a black-box testing method
- Focuses on requirements testing
- Can view testing as a search problem
- Learning-based testing is a search heuristic
- Guided by analogies between learning and testing

TCG for a Reactive System:
Coffee Machine

Sys-Req: always(in=$1 implies after(10, out=coffee)) *pass/fail*



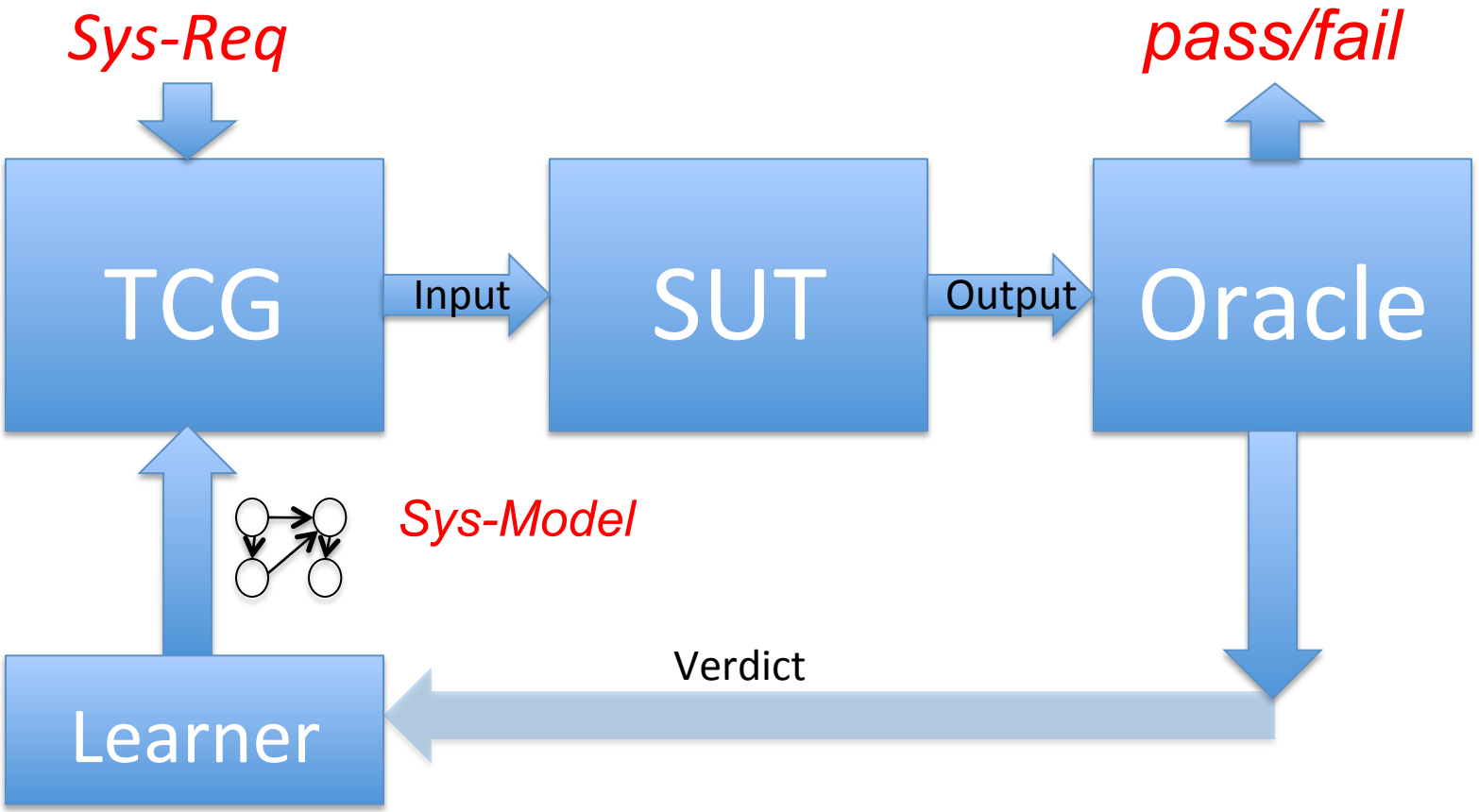
$in_0 := \$1, out_{11} := coffee$ Satisfies
always(in=1$ implies after(10, out=coffee))

Key Problem: Feedback

Can we modify this TCG architecture to..

1. Iteratively improve the next test case
2. Execute a large number of tests?
3. Obtain good coverage?
4. Find bugs quickly?

Learning-Based Testing



"Model based testing without a model"

Basic Idea …

LBT is a search heuristic that:

1. Incrementally learns an SUT model
2. Uses generalisation to predict bugs
3. Uses best prediction as next test case
4. Refines model according to test outcome

Abstract LBT Algorithm

1. Use $(i_1, o_1), \ldots, (i_k, o_k)$ to learn model M_k
2. Model check M_k against *Sys-Req*
3. Choose "best counterexample" i_{k+1} from step 2
4. Execute i_{k+1} on SUT to produce o_{k+1}
5. Check if (i_{k+1}, o_{k+1}) satisfies *Sys-Req*
   a) Yes: terminate with i_{k+1} as a bug
   b) No: goto step 1

Difficulties lie in the technical details …

Preliminary Conclusions

- A promising approach …
- Flexible general heuristic,
  - many models and requirement languages seem possible
- Many SUT types might be testable
  - procedural, reactive, real-time etc.

Course conclusions

- Testing is a big field
- Testing is an old field: techniques from 1950s
- Testing is a new field: model checkers, etc.
- We have focused on:
  - Glass box
  - Black-box
  - Requirements
  - Model-based
  - ATCG

Major Omissions

- OO testing
- Interface testing
- Integration testing
- Agile testing

You can teach yourself many of these topics from the course textbook, or other sources!

Testing is a subject in progress!