Test Automation

20 December 2017

The problem of test automation

Testing has repetitive components, so automation is justified

The problem is *cost-benefit evaluation* of automation [Kaner]

Time for: test creation, checking their functionality, documentation

Is automation reusable? (if the program evolves)

Is maintenance needed? (GUI change, internationalization)
⇒ *Test automation: treated like any software development*

Does it delay finding bugs? (fewer resources to run tests)

Does it find enough bugs? Or are most found by manual testing

Is it powerful enough? Or does it automate only “easy” tests?

Example: Capture-replay

1) Record user actions (*mouse/keyboard*) and resulting screen (bitmap) ⇒ most primitive level
– other checks: with tester effort (interrupt/insert)
– fragile: susceptible to any product change
– possible comparison errors in resulting image

2) script with *high-level actions* (select menu/button)
– more flexible, but does not check graphic layout (low level: font, text size/overwrite, etc.)

3) *scripting language* to automatically generate new tests

Disadvantage of capture-replay
Cannot continue from errors
⇒ errors are found manually in the recording process
⇒ only rerunning a "good" test is automated (regression)
Does not define tests *implicit* for human ("all the rest is OK") (cannot detect unspecified errors, is inflexible – e.g. bitmap)

Example: Test monkeys

Automated tools that execute random tests
(without a testor's knowledge on product functionality)
Dumb monkeys: completely ignore purpose (know just mouse/keyboard)
  but may have basic notions about windows/menus/buttons
Smart monkeys: have a *state model* of the application, explore transitions between these states

++ can sometimes find 10-20% of errors [Nyman, Microsoft, 2000]
++ good preliminary coverage (e.g.: 65% in 15 min for a text editor)
++ completely automated, no human effort for test capture
-- "dumb": only bug known to monkeys is system crash
-- ⇒ errors are hard to record and reproduce
++ runs independently, unsupervised, minimal resources (cost)

What can we automate in testing?

Test execution

e.g. any unit testing framework
useful in

Test evaluation

= problem of *test oracle* : did the test pass ?
Nontrivial, often needs manual inspection. Risks:
– undetected errors (imprecision)
– false warnings ⇒ cost of manual checking
Ex: compare continuous signals (in automotive industry)
image comparison (for screen/printer)

Test generation

Relatively easy: generating test skeletons (declarations + calls)
More difficult: intelligent generation of relevant data (coverage)

Choosing a test architecture [Kaner]

1) *Data-driven* architecture
   separates data from test structure (like in programs)
   Example: table. row = test; columns = test parameters
   A script generates a test case for every table row
   Minimal reasonable coverage: every *pair* of parameter values
(for every *combination* of values, number is exponential)

2) *Framework-based* architecture
   A library of functions separates testing from UI
   e.g. `open(`$file$`)`, independent of actions for opening
      (menu, button click, keyboard, etc.)
   ++ reuse for frequent actions
   ++ indirection $\Rightarrow$ insulation from testing tool
   -- costly, amortized only in future releases

Specification-based testing

Automatable (*keyword testing*) for spects in well-defined language
Starting from documentation: tabular spec, e.g. [Pettichord]

Test ID	Operation	Table	Name	Type	Nulls
dtbed101	Add Col	TB03	NEW_INT_COL	CHAR(100)	Y

Important: choose format easily understandable by user
A translator/test interpreter generates the test driver from the table or interfaces with the (commercial) testing tool used
++: requirement-driven (*what*, not *how*), independent of implementation and testing tool, self-documented

More advanced: automated test generation from specs in formal language

- e.g. decision tables in RSML in TCAS-II aviation protocol
- test generation from timing diagrams in embedded systems

Model-based testing

Models: finite automata, UML, Statecharts (hierarchical automata), Message Sequence Charts, timed automata, Petri nets, Markov chains...

Test generation criteria: satisfactory model coverage
all states / transitions; combinations of k consecutive transitions (*k-switch cover*)

++ facilitates generation of relevant tests
−− investment in model building and maintenance

Testing based on *model checking*
by *state space exploration*, starting from specifications:
1) question: can the model reach a given state ?
2) if so, a *model checker* will generate an example trace = test case

Implementation-based testing: symbolic execution

Goal: exercising program, satisfying a *coverage criterion*
$\Rightarrow$ needs: instrumentation to measure test coverage
How: set of *paths*: random choice + directed search
(to reach branches not yet covered)

Symbolic execution: executing program using *expression*
with symbolic variables, rather than concrete (numeric) values

Symbolic execution gathers *path conditions* for followed branches

Satisfiability of conditions is checked with specialized tools
(satisfiability checkers, constraint solvers)
$\Rightarrow$ generate input data that will exercise that path
or prove path is infeasible $\Rightarrow$ stops exploring that path

Symbolic execution for program testing

described as early as 1976 (James C. King)

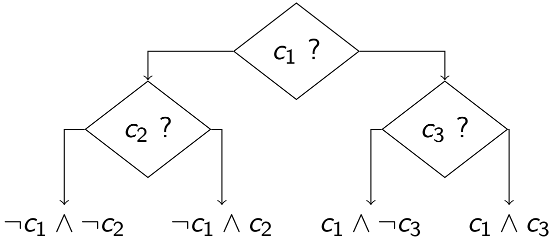
program is executed by a special interpreter, using *symbolic* inputs
$\Rightarrow$ results in symbolic execution tree
    tree traversal stops when path condition becomes unsatisfiable

Test generation purpose:
  attaining high coverage
  sometimes, reaching a specific branch

Successful mature technique, hundreds of papers, many tools:
Java Pathfinder, (j)CUTE, Crest, KLEE, Pex, SAGE, ...
  for C/C++, C#, Java, more recently JavaScript

Variants of symbolic execution

Classic *completely symbolic execution*
explores each execution path independently



Problem: must express all program/language semantics as formula
solving arbitrary formulas impossible (limited to simple arithmetic)
reality: complex math, library function, environment
solution: *model* libraries & environment
e.g. KLEE tool has models for some 40 syscalls (2.5 kloc)

Dynamic (concolic) symbolic execution

symbolic execution is directed by *concrete* run (hence: "conc olic"

When symbolic execution is infeasible, perform a concrete execution step

e.g. nonlinear arithmetic, library/system functions

```
**function** explore($pathcond = [c_1, c_2, \ldots, c_n]$)
**for** $k = n$ downto 1 **do**
    inputs = solve $pathcond = c_1 \wedge \ldots \wedge c_{k-1} \wedge \neg c_k$    (flip $c_k$)
    rerun with new inputs; capture new pathcond'
    explore(pathcond')
```

Problem: by using concrete values, might not reach desired path

Concretization as potential obstacle

```
y = hash(x);      // can't solve hash formula => y is
if (x + y > 0)
   // path 1
else
   // path 2
```

Assume: x = 20; y = hash(20) = 13 ⇒ *path 1*
To reach *path 2*, negate $x + y > 0$, with *concrete* y (constant 13)
Solver might return, e.g., x = -15
but we might have hash(15) = 27 (can't predict) and then
$x + y > 0$
  ⇒ execution still follows path 1

⇒ *retry*; worst-case: degrades to *random testing*

When is test automation efficient ?

In *regression testing*: need only store tests and expected results (and means to automate comparison)

Testing user interfaces (discussed earlier)

Testing compilers / translators
  automated test generation starting from input grammar
  explores random/statistic combination of grammar rules

Load/stress testing: random; quantity rather than content is relevant

Fuzz testing: generate large quantities of random / possibly hostile input, to detect input validation errors or security vulnerabilities

e.g. RANDOOP [Microsoft]: 4 M tests in 150 CPU hours / 15 person-hr
30 bugs in code tested for 200 person-years, vs. 20 errors/year manually
see also `http://research.microsoft.com/en-us/projects/Pex/`

Basic workings of a fuzzer

e.g. American Fuzzy Lop `http://lcamtuf.coredump.cx/afl/`

- maintains queue of test inputs
- mutates inputs using several strategies
- if new coverage achieved, add mutant to input queue
- minimize each test input (keeping coverage)
- minimize input corpus (avoids overlap)

records *transition coverage* between program basic blocks
classifies runs into crashes/hangs/normal exit

highly successful, found many security vulnerabilities

mutating inputs can synthesize interesting formats (e.g. images)

can identify format fields with various meaning
  (length, checksum, payload, control opcode, etc.)

Automated debugging

After automating detection $\Rightarrow$ help in *fault localization*

Minimizing test inputs
  binary search, finds (file) input half that caused error

Minimize differences between correct and erroneous run
  also binary search, for two close inputs

Fault localization *in space*
  in debugger, compare execution state of correct and buggy run
  detect (precisely/statistically) invariants/patterns violated by erroneous run

Fault localization *in time*
  compare erroneous runs and find points where *infected* variables start affecting output

Delta debugging [Zeller]: partial automation of these techniques