Software Testing and Validation

A.A. 2025/2026

Corso di Laurea in Informatica

Testing Preliminaries

Igor Melatti

Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

From Formal Verification to Testing

- Main analogies: both formal verification and testing are about checking some properties of a system
  - easiest property: does the system output the correct answer for a given input?
  - other properties: does it deadlock? does it run within given deadlines?
- Main difference: formal verification requires a formal *model* of the system and a specification of the properties in some temporal logic
  - in some cases, the model can be automatically built (e.g., for hardware verification or bounded software verification)
- Testing requires the current version (for part) of the *actual* software
  - as for the property, no need that any temporal logic is used, though it may help
  - a simulator may be used for some physical components

From Formal Verification to Testing

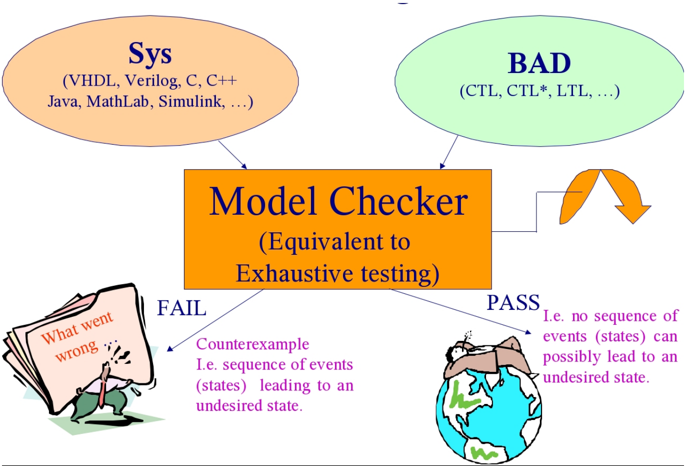
- Thus, testing is typically applied *late* in the design process
  - you need actual software, which is typically developed after architectural design and so on
  - at least for complex software projects
- However, if the software design process is well organized, testing may also be applied much early
  - e.g.: some components may be fully developed before others
  - as soon as they are developed, they may be tested
  - this is actually what it should be *always* done
  - the technique allowing this is called *scaffolding*

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

From Formal Verification to Testing

- So, no models in testing? NO!
  - you may not have a model of the system itself, but models however play an important role
  - in some cases, also a model of the system is available, why not to use it?
- Models in testing are typically used:
  - to generate inputs
  - to guide in generating inputs
  - to undestand if a testing phase is "adequate" or not
- What about algorithms?
  - no "real" algorithms are used in testing
  - forget μ-calculus or nested DFS or so on
  - though, as we will see, some algorithms may be helpful, exactly as for the models

From Model Checking...



An approximate answer
BUG HUNTING: Testing + Simulation

Input sequence (stimulus)

... **u**(3) **u**(2) **u**(1) **u**(0)

System (Model)

Compute output by Simulation or by running the actual system when possible

Output sequence

y(0) **y**(1) **y**(2) **y**(3) ...

Define initial state + parameters

Observer
checks that output sequence ok

Basic Notions on Testing

- No need of complex algorithms as in model checking: simply
  1. choose what to test
     - SUV: System under Verification
  2. devise some relevant inputs
  3. execute the SUV with such inputs
  4. check if the corresponding results are ok or not
- Some automatization may be performed, but mainly a manual work for each of such steps
- Does this mean testing is easy? Obviously, NO!

Basic Notions on Testing

- Main difficulties for "choose what to test"
  - only the whole system?
    - for cyber-physical systems, a simulator may be used
  - all/some meaningful parts?
  - all/some functionalities?
  - all/some single functions or classes?

Basic Notions on Testing

- Main difficulties for “devise some relevant inputs”
  - are there some “hidden” inputs?
    - e.g., a global/class member variable, a database or a RESTful querying system
  - which are the input variables domains?
  - which values have to be selected?
    - variable domains may be extremely large (integers) or close to infinite (floats, strings)
    - sometimes, also go outside variables domains
  - once values are selected, may we understand which share of the input space got covered?
  - all inputs at the beginning, or must be fed with some timing?

Basic Notions on Testing

- Main difficulties for “execute the SUV with such inputs”
  - whole system: sometimes easy, sometimes hard
    - a simple Web interface and a submit button: easy
    - sequence of commands in CLI
    - only invokable by some mouse clicks sequence (hidden functionality)
    - many library requirements
    - ...
  - only some part(s): must be extracted in some way for execution
    - all dependencies must be resolved
  - all inputs at the beginning, or must be fed with some timing?

Basic Notions on Testing

- Main difficulties for "check if the corresponding results are ok or not"
  - in many cases (usually, not for the whole system...), the result isn't easy to extract
    - it may be in some global variable, needs some extra code to output its value after the function call
    - or in a private class member without a getter: the getter must be added (or the private removed)
    - the result may be put in some file/database/socket
  - but also after we got the result: how to know if it is correct? isn't it a circular reasoning?
  - if we simply want to detect failures, it may be needed to modify the code to expose them
    - segmentation faults for C/C++ code...
  - check for software usability: what is "ok" is not well defined...

Basic Notions on Testing

- No general tool is available for none of the problems above
  - some subparts may be automatized for many interesting cases, but not for all cases
- Testing must be integrated within software process
  - only testing at the end of the developing phase is bad
- Running tests has a cost: consider project budget and release deadlines
  - may force to only check the whole system with very limited time

Basic Notions on Testing

- Model Checking is only performed for mission- or safety-critical systems with medium-high budget
- Testing is *always* performed on *any* software
  - from cli-based computer-science-first-year projects to airport management system
- Even more: any released version of a software must be tested
  - otherwise, CrowdStrike-like problems
  - easy to understimate the effect of a modification on a working software
  - Regression Testing

Basic Notions on Testing

- Testing all features is typically impossible for complex projects
- For simpler projects, all features may be tested, but not at 100% coverage
  - actually, also 1% testing coverage is often too difficult to obtain
  - even if there are some particular measures for which 100% coverage may be achieved
- Priorities must be defined
  - more risky and defect-likely parts of the software must be checked first
  - strictly followed by the parts which are executed more often
  - 80/20 rule: 80% of any daily business activity is provided by 20% of the business system functions, transactions, or workflow

Basic Notions on Testing

- Some authors distinguish between *positive* vs *negative* testing
  - positive testing: check that the software behaves as it should under "normal" conditions
  - negative testing: check that users cannot "break" the system
    - both if they are malicious: denial of service and similar issues
    - and if they are not: including apex by accident in a non-sanitized search...
- We will consider both these types of testing without further distinction

Basic Notions on Testing

- Software related figures:
  - architects: general software “structure”
  - developers: write (and fix!) code
  - testers: plan, prepare and execute tests; furthermore, interpret test results
- For not-too-big projects, some or all of these figures could coincide
- For complex projects, testers should be separated from developers and/or architects
  - though they must know the corresponding skills
  - not influenced by having devised and/or implemented the system
  - the same programmer could be a developer in a project and a tester in another

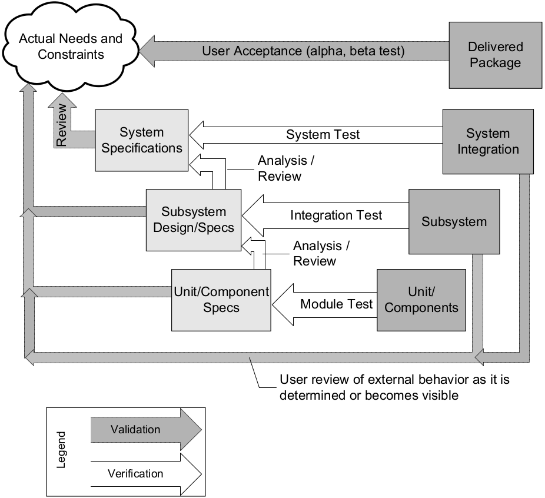
Basic Notions on Testing

- Skills of a software engineer specialized in testing: matches the steps from above
  - choose what to test
  - devise some relevant inputs
  - execute the SUV with such inputs
  - check if the corresponding results are ok or not
- All keeping into account budget (i.e., number of testering engineers) and deadlines (i.e., time available for testing)
- In a testing team, such capabilities could be properly mixed

Testing Timeline

- Let us consider complex projects: the following types of testing can be performed
  1. *unit testing*: test simple functions/classes/processes first
  2. *integration testing*: put some meaningful subsets of functions/classes/processes together and test them
  3. *system testing*: test the whole system
     - last step of integration...
  4. *acceptance testing (validation)*: test the whole system with the final users
  5. *regression testing*: how to re-test the system when new releases are issued
     - code (and possibly specifications) is modified

Testing Timeline



Testing Timeline

- Some of these steps may be deleted
  - for cli-based computer-science-first-year projects, unit testing is enough
  - for medium-size projects, integration testing and system testing may coincide
  - for a personal software, validation is straightforward as developers and final users coincide
- Not necessarily in cascade
  - errors discovered in later steps typically cause earlier steps to be re-run
  - sometimes not only re-running, but also devising new inputs could be required
- If errors are discovered, developers have to fix them; then, re-run testing

Testing Timeline

- It is easy to understand why you need regression testing
- Why not directly acceptance testing?
  - going to the final user with a non-working project is obviously a bad idea
  - without testing, your software may simply don't run at all
- Why not directly system testing?
  - same as asking "why do you use software engineering techniques to write complex software?"
  - testing is bottom-up (from units to system) because:
    - it is far easier for developers to understand where errors are
    - allows reusability of units

Testing Main Techniques

- Two main overall methodologies:
  - *functional testing*: tester knows specs but not the code
    - also known as *black-box testing*
  - *structural testing*: tester exploits code knowledge
    - also known as *white-box testing* and *glass-box testing*
    - includes *data-flow testing*
- Two other methodologies
  - *static testing*: code may not exist, look at documentation only
    - mainly done through model checking
    - but also prototyping may be used
  - *performance testing*: needs actual code
- Applicable to all types of testing, from unit to acceptance

Testing Main Techniques

- Orthogonal and/or auxiliary techniques:
  - *combinatorial testing*
    - given some values for single inputs, obtain a full input
  - *model-based testing*
    - extract inputs from models of software
    - special case: *fault-based testing*
  - *test execution*: not always straightforward
- Applicable to nearly all testing of the previous slide

Basic Notions on Testing

- Testing is not only for software: nearly all products must be tested before being sold
  - i.e., stressed in a controlled environment
- Typically, the testing phase is standardized for a given product
  - always repeated for some randomly chosen instance of the product
  - e.g., take a smartphone from a selling pack and drop it from 10m
- For products which are not built in series, testing must be individual
  - race cars, houses, etc.
- Of course, some guidelines may be available
  - e.g., testing of houses in a seismic environment

Basic Notions on Software Testing

- Software is among the most difficult things to be checked
  - it is virtually always "customized", thus each software needs its own testing phase
- There are guidelines, some of which will be covered in this course
- Some difficulties:
  - only errors presence can be proved
  - cost
    - it is easy to make some simple tests
    - it may be enough for very-non-critical software
    - for most software, a tradeoff is needed between testing cost and software criticality

- Some difficulties (continued):
  - non-linearity
    - if you successfully test an elevator to be able to carry 1000 kg, then it will be ok with 900 kg or less
    - if you successfully test a sorting procedure with 1000 elements, it may fail with 2 elements
    - if you make a small modification to a pair of glasses, you do not need to run full design test from scratch
    - if you make a small modification to a software (e.g., a security update), it may cause some failure in other previously tested parts of the software
    - recall the CrowdStrike vulnerability...

Six Principles for Testing (and Verification)

- The following principles characterize the Testing and Validation as an activity of its own:
  - partition, visibility, feedback
    - nothing new: also other engineering activities may use this
  - sensitivity, redundancy, restriction
    - specific for testing and validation

Six Principles for Testing (and Verification)

- *Sensitivity*
  - problem: many errors may not be "observable"
  - e.g., a buffer overflow in C/C++ may or may not cause a failure in the running process
  - sensitivity asks that errors or faults in the software always result in observable failures
  - especially hits in code design/implementation: add assertions or similar code fragments
    - or use languages with dynamic checks such that Java, Python or Rust
  - as for verification, model checking is actually more suited for sensitivity
  - also manual code inspection may be used

Sensitivity Example

```c
/**
 * Worse than broken: Are you feeling lucky?
 */

#include <assert.h>

  char before[ ] = "=Before=";
  char middle[ ] = "Middle";
  char after[ ] = "=After=";

void show() {
  printf("%s\n%s\n%s\n", before, middle, after);
}

void stringCopy(char *target, const char *source, int howBig);

int main(int argc, char *argv) {
  show();
  strcpy(middle, "Muddled");      /* Fault, but may not fail */
  show();
  strncpy(middle, "Muddled", sizeof(middle)); /* Fault, may not fail */
  show();
  stringCopy(middle, "Muddled",sizeof(middle)); /* Guaranteed to fail */
  show();
}

/* Sensitive version of strncpy; can be counted on to fail
 * in an observable way EVERY time the source is too large
 * for the target, unlike the standard strncpy or strcpy.
 */
void stringCopy(char *target, const char *source, int howBig) {
  assert(strlen(source) < howBig);
  strcpy(target, source);
}
```

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Six Principles for Testing (and Verification)

- *Redundancy*
  - in a broad sense: having some behavior that depend on something other
  - you declare an 'intent", so we can test if the intent is fulfilled
  - typed languages are a type of redundancy by intent
    - e.g., you declare something to be integer and you can raise an error if instead there is a float
  - as for actual testing: check if an implementation is ok w.r.t. its specification is actually a type of redundancy
  - specifications should be written so as to ease automatic testing or manual inspection

Six Principles for Testing (and Verification)

- *Restriction*
  - your desired property is too difficult to attain?
  - restrict it, i.e., try with something easier
    - but however meaningful
  - e.g.: too difficult to check that a variable is always initialized, then check if there exist the possibility that it may be uninitialized

Restriction Example

```
/** A trivial method with a potentially uninitialized variable.
  * Maybe someCondition(0) is always true, and therefore k is
  * always initialized before use ... but it's impossible, in
  * general, to know for sure. Java rejects the method.
  */
static void questionable() {
    int k;
    for (int i=0; i < 10; ++i) {
        if (someCondition(i)) {
            k = 0;
        } else {
            k += i;
        }
    }
    System.out.println(k);
}
}
```

Six Principles for Testing (and Verification)

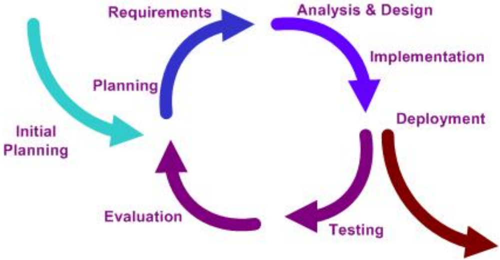
- *Partition*
  - divide and conquer (divide et impera)
  - decompose the problem to be tested
  - the very fact that many different testing techniques exists, and may be employed on the same software, it is a matter of partition
    - unit testing, functional testing, structural testing...
  - also making a model of the system is a partitioning technique
    - from "does this software satisfy my property?"...
    - to "does this model satisfy my property?" and "does this model faithfully represent the software?"

Six Principles for Testing (and Verification)

- *Visibility*
  - very similar to sensitivity, but with focus on input rather than output
  - again, mainly a design issue to ease testing
  - typical example: base program information on textual files rather than binary files
    - low performance degradation, but much better readability and capability of testing
  - e.g., HTTP exchange information as text
  - e.g., Unix-based OSs use text files for configuration
- *Feedback*
  - learn to build better testing phase from previous testing phase

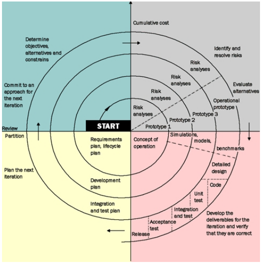
Software Process

- Not "process" in the sense of operating systems: "software process" is the whole set of activities needed to develop a high-quality software for some specific problem
  - software process contains: requirement analysis and specification, software design, implementation, validation and verification
  - organized in many ways
- Testing (and verification in general) cannot be simply done at the end



Software Process

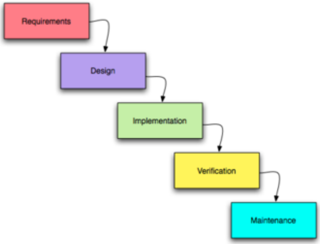
- Not “process” in the sense of operating systems: “software process” is the whole set of activities needed to develop a high-quality software for some specific problem
  - software process contains: requirement analysis and specification, software design, implementation, validation and verification
  - organized in many ways
- Testing (and verification in general) cannot be simply done at the end



UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

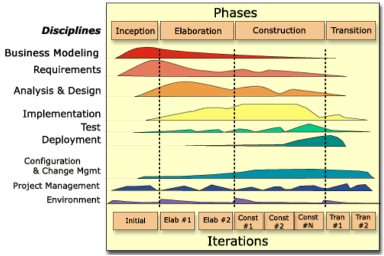
Software Process

- Not "process" in the sense of operating systems: "software process" is the whole set of activities needed to develop a high-quality software for some specific problem
  - software process contains: requirement analysis and specification, software design, implementation, validation and verification
  - organized in many ways
- Testing (and verification in general) cannot be simply done at the end



Software Process

- Not "process" in the sense of operating systems: "software process" is the whole set of activities needed to develop a high-quality software for some specific problem
  - software process contains: requirement analysis and specification, software design, implementation, validation and verification
  - organized in many ways
- Testing (and verification in general) cannot be simply done at the end



Software Process: Testing

Completeness important class of faults are suitably targeted

- "important" depends on what you are building
- e.g., if C/C++ is used, beware of memory leaks

Timeliness discover errors as soon as possible

- error in coding revealed at unit testing OK
- error in coding revealed at system integration BAD
- error in coding discovered by final user VERY BAD
- error in the system specifications discovered in system acceptance test CATASTROPHE

Cost effectiveness achieve completeness and timeliness within budget

- on the whole process: do not repeat heavy tasks because of errors

Software Quality Through Testing

- Process visibility: progress must be easily detectable
- This entails that quality goals must be clearly stated and refined
- Goals are measured on software product qualities, which may be:
  - *internal*: only visible to the software developers and designers
    - e.g.: maintainability, reusability, traceability
  - *external*: also visible to final users
    - e.g.: throughput, latency, usability
    - summing up, either dependability or usefulness goals
    - dependability: does it have (functional) faults?
    - usefulness: provided it is dependable, does it have other (typically non-functional) faults?
    - e.g.: bad user interface, software is too slow, etc

Software Dependability

- Simplest dependability property: correctness
  - all behaviors of the software are as specified
- Reliability: statistical approximation of correctness
  - if not all behaviors are ok, then at least, e.g., 90% of them are
  - often specified w.r.t. a particular usage profile
  - the same program can be more or less reliable depending on how it is used
  - a possible formal definition: percentage of successful operations in a given period $\frac{100|S|}{|S|+|F|}$
    - S is the set of all operations which succeed in the given period

Software Reliability: Other Possibile Definitions

- Availability: reliability w.r.t. failures duration is important
  - may be defined as $100\frac{u}{u+d}$
  - u: software is up and accepting requests
  - d: software is down and not accepting requests
  - typically, $u + d = 1$ day, or 1 week
- MTBF: Mean Time Between Failures
  - may be defined as $\frac{1}{|F|}\sum_{f \in F} |f|$
  - F is the set of all failures in the given period (1 day, 1 week...)
  - for a failure $f \in F$, $|f|$ is the duration, i.e., time required for fixing f
  - more detailed than availability: e.g., it distinguishes from 30 failures of 1 minute and 1 failure of 30 minutes

- Robustness: correct and reliable only within some defined operational limits
  - if there is a failure only because of a 100x load, the system is however robust
- Safety: nothing bad occurs
  - of course, must be defined w.r.t. some property
  - e.g.: there is never more than one process in the critical section
  - broader sense than what we have defined in model checking
  - essentially, it is any property you can specify, so also liveness and neither liveness nor safety

Terminology

Program or System Under Verification (SUV)

- could also be a part of a "program"
- could also be a system with many processes

Test case A set of inputs, execution conditions, PASS/FAIL criterion

- input is anything the program to be tested can get
  - command-line arguments, files, interrupts, mouse coordinates, sensors...
- execution condition: information on the test execution
  - typically, input timing: whether all input must be provided at the start or not
  - e.g., a sequence of interrupts with given timing
- PASS/FAIL: some way to check
  - e.g.: output must be equal to this expected result

Terminology

Test case specification A formal or informal description of a test case

- "the input is two words" $\rightarrow$ a valid test case will be "goodbye all"

Test suite a set of test cases

Test execution running the test cases on the program

Test obligation a property for test case specifications

- e.g., "all words must be 7 letters long"

Adequacy criteria some property a test suite must fulfill

- e.g., "all test cases must contain at least 30 inputs"
- could also be seen as a set of test obligations
- namely, the adequacy criterion is satisfied if every test obligation is satisfied by at least one test case in the suite

Terminology

Function Mathematical concept (set of pairs)

Java Function Syntactical function in Java language

- works with all other languages, of course

Unit Smallest unit of work in the program

- typically (but not always) close to single functions or single classes
- here, “unit of work” roughly refers to:
  - the smallest increment by which a software system grows or changes
  - the smallest unit that appears in a project schedule and budget
  - the smallest unit that may reasonably be associated with a suite of test cases (*unit testing*)

Terminology

Independently Testable Feature (ITF) Some functionality of the program which can be isolated from the other functionalities

- not necessary at code level: here, it is testing level
- e.g., a program or a function may be able to both sort and merge files
- however, sorting and merging may be ITF
- granularity depends on the program: from individual functions, to features of an integrated system composed of many programs
- going through individual classes and libraries
- when detected at unit testing, an ITF is usually a function/method or a class, but not only unit testing exists...