Verification & Validation

Dr. Guillaume Brat
NASA Ames Research Center

Contents

• Motivation
  – Ariane 501
  – Space Examples
  – System complexity
• Basic Definitions
  – Verification vs. Validation
  – V&V and Process
• Traditional Techniques
  – Audits, reviews, inspections
  – Dynamic verification (testing)
    • Levels
    • Basics
    • Terminology
    • Mutation testing
    • Coverage
  – Static verification
• Formal Analysis
  – Static analysis
  – Model checking
Contents

• Motivation
  – Ariane 501
  – Space Examples
  – System complexity
• Basic Definitions
  – Verification vs. Validation
  – V&V and Process
• Traditional Techniques
  – Audits, reviews, inspections
  – Dynamic verification (testing)
    • Levels
    • Basics
    • Terminology
    • Mutation testing
    • Coverage
  – Static verification
• Formal Analysis
  – Static analysis
  – Model checking

Ariane 501

https://www.youtube.com/watch?v=gp_D8r-2hwk
Cost of Failure Examples in Space

$165M

$125M

>M$1B

Mars Missions Bugs

- Mars Path Finder:
  - Code size: 140 KLOC
  - Famous bug: priority inversion problem

- Deep Space One:
  - Code size: 280 KLOC
  - Famous bug: race condition problem in the RAX software

- Mars Exploration Rovers:
  - Code size: > 650 KLOC
  - Famous bug: Flash memory problem
Increasing Software Complexity

**SW size (MLOC)**

Contents

- **Motivation**
  - Ariane 501
  - Space Examples
  - System complexity

- **Basic Definitions**
  - Verification vs. Validation
  - V&V and Process

- **Traditional Techniques**
  - Audits, reviews, inspections
  - Dynamic verification (testing)
    - Levels
    - Basics
    - Terminology
    - Mutation testing
    - Coverage
  - Static verification

- **Formal Analysis**
  - Static analysis
  - Model checking
Verification vs. Validation

- **Verification**
  - Are we building the product right?
  - The system conforms to its specification

- **Validation**
  - Are we building the right product?
  - The system should solve the right problem

The V & V process

- V & V must be applied at each stage in the software process.

- It has two main objectives
  - Discovery of defects in a system;
  - Assessment of whether or not the system is useful and useable in an operational situation.
Process and V&V

Contents

• Motivation
  – Ariane 501
  – Space Examples
  – System complexity
• Basic Definitions
  – Verification vs. Validation
  – V&V and Process
• Traditional Techniques
  – Audits, reviews, inspections
  – Dynamic verification (testing)
    • Levels
    • Basics
    • Terminology
    • Mutation testing
    • Coverage
  – Static verification
• Formal Analysis
  – Static analysis
  – Model checking
Audits, Reviews, and Inspection

• V&V use these techniques to verify the software during its development process
  – Peer Reviews
  – Documentation inspections
  – Requirements/design/code reading
  – Test witnessing
  – Installation audits

Dynamic Verification

Dynamic verification is performed during the execution of software, and dynamically checks its behavior

• Unit Testing: executed at code level.
  – Unit testing eliminates bugs at an early stage, though all defects cannot be uncovered by unit testing.
• Integration Testing: associated with the architectural design phase.
  – Integration tests are performed to test the coexistence and communication of the internal modules within the system.
• System Testing: associated with the System design phase.
  – System tests check the entire system functionality and the communication with environment.
• Acceptance Testing: associated with the delivery phase
  – Acceptance testing involves testing the product in user environment and uncover compatibility issues and non functional issues (load and performance defects) in the actual user environment.
Levels of Testing

- What users really need
- Requirements
- Design
- Code
- Maintenance

Acceptance testing
System testing
Integration testing
Unit testing
Regression Testing

Testing basics

- Test Cases
- Testing
  Execution of the code based on the test cases
- Test Outputs
Testing terminology

- Black Box testing: no knowledge of the implementation
- White box testing: full knowledge of the implementation

Test cases

Test cases can be generated
- Manually
- Automatically
Mutation testing

• Create a number of mutants, i.e., faulty versions of program
  – Each mutant contains one fault
  – Fault created by using mutant operators

• Run test on the mutants (random or selected)
  – When a test case reveals a fault, save test case and remove mutant from the set, i.e., it is killed
  – Continue until all mutants are killed

• Results in a set of test cases of high quality

Mutation Testing

• Mutants

```c
int getMax(int x, int y) {
    int max;
    if (x > y)
        max = x;
    else
        max = y;
    return max;
}
```
Testing Terminology

Black box testing

White box testing

visibility into the code

Pass/Fail?

It is very often hard or time-consuming to determine manually if a test passes or fails

Testing Oracles

Black box testing

White box testing

visibility into the code

Pass/Fail?

Test oracles automate the process of finding out the result of tests
When is it enough?

When do I know that I have done enough testing?
Test coverage

Coverage

- **Statement coverage**
  - Each statement should be executed by at least one test case
  - Minimum requirement

- **Code coverage strategies, e.g.**
  - Decision coverage
    - All paths should be executed by at least one test case
    - All decisions with true and false value
  - Path coverage
  - Data-Flow analysis (Defines -> Uses)
  - Specification-based testing, e.g.
Coverage

• Specification-based testing, e.g.
  – Equivalence partitioning
    • One witness by equivalence classes
  – Boundary-value analysis
    • Stressing boundary conditions
  – Combination strategies
  – Requirement-based coverage
    • Generate test cases based on the requirements
    • Can also use design models

Static verification

Static verification is the process of checking that software meets requirements by inspecting the code before it runs

• Code conventions verification
• Bad practices detection
• Software metrics calculation
• Formal verification, e.g.,
  – Static analysis
  – Abstract interpretation
  – Software model checking
  – Pre- and post-condition analysis

Most used in practice

Is becoming more common

Still mostly research topics
Contents

• Motivation
  – Ariane 501
  – Space Examples
  – System complexity
• Basic Definitions
  – Verification vs. Validation
  – V&V and Process
• Traditional Techniques
  – Audits, reviews, inspections
  – Dynamic verification (testing)
  • Levels
    • Basics
    • Terminology
    • Mutation testing
    • Coverage
  – Static verification
• Formal Analysis
  – Static analysis
  – Model checking

Why formal verification?

Cost of fixing Software Bugs


Increasing Software Complexity
Where can we use Formal Methods?

Ariane 501: the bug

- They re-use code from Ariane 4
  - Some of the exception handlers were “optimized away” from the Ariane 4 code base for performance reasons
  - Optimizations were forgotten during re-use for Ariane 5
- Some value for a power load was much bigger in Ariane 5 than Ariane 4
  - It caused a numerical overflow
  - The exception handler was missing
  - Exception wasn’t caught
Static Analysis

Static analysis offers compile-time techniques for predicting safe and computable approximations to the set of values arising dynamically at run-time when executing the program. The analysis is done without executing the program, and all possible values (and more) are computed.

For example, abstract interpretation techniques work by extracting a safe system of semantic equations which can be resolved using lattice theory techniques to obtain numerical invariants for each program point.

Static analysis example

- Static analyzers check that every operation of a program will never cause an error (division by zero, buffer overrun, deadlock, etc.)
- Example:

  ```c
  int a[1000];
  for (i = 0; i < 1000; i++) {
    safe operation ———> a[i] = ... ;  // 0 <= i <= 999
  }

  buffer overrun ———> a[i] = ... ;  // i = 1000;
  ```
Defect Classes covered by static analysis

- Static analysis is well-suited for catching runtime errors, e.g.:
  - Array-out-bound accesses
  - Un-initialized variables/pointers
  - Overflow/Underflow
  - Invalid arithmetic operations
- Defect classes for Deep Space One:
  - Misuse: array out-of-bound, pointer mis-assignments
  - Initialization: no value, incorrect value
  - Assignment: wrong value, type mismatch
  - Undefined Ops: FP errors (tan(90)), arithmetic (division by zero)
  - Omission: case/switch clauses without defaults
  - Scoping Confusion: global/local, static/dynamic
  - Argument Mismatches: missing args, too many args, wrong types, uninitialzed args
  - Finiteness: underflow, overflow

Mars Missions Bugs

- Mars Path Finder:
  - Code size: 140 KLOC
  - Famous bug: priority inversion problem
- Deep Space One:
  - Code size: 280 KLOC
  - Famous bug: race condition problem in the RAX software
- Mars Exploration Rovers:
  - Code size: > 650 KLOC
  - Famous bug: Flash memory problem
Model Checking

Model checking performs an exhaustive search of the reachable state space of a system checking whether certain behavioral properties hold. It allows detection of:

- Generic properties
  - assertion violations, deadlocks, race-conditions, uncaught exceptions
- Application specific properties
  - program invariants, pre/post-conditions - could include temporal logic properties

Analysis of Models

• individual state machines for subsystems / features
• automatically combines behavior of state machines
  - exhaustively explores all executions in a systematic way
  - handles millions of combinations – hard to perform by humans
  - reports errors as traces and simulates them on system models
Program Model Checking vs. Testing

```
void add(Object o) {
    buffer[head] = o;
    head = (head+1)%size;
}

Object take() {
    tail = (tail+1)%size;
    return buffer[tail];
}
```

**Improves testing:**
- Ability to control thread scheduling & environment responses.
- Stress test critical system states.
- Directed search for specific errors: deadlock, race conditions, assertion violations

Compositional Verification

- The goal is to achieve scalability of formal techniques to large software systems.
- Use system's natural decomposition into components to break-up the verification task
  - Divide-and-Conquer approach
- Components typically satisfy requirements in specific contexts / environments
  - safety assumptions about contexts
- System safety derives from the ability to compose the components' contexts at the system level
Sometimes the bugs are not in the Software

- At 12:12 GMT 13 May 2008, a NASA Space Shuttle was loading hypergolic fuel for mission STS-124 when a 3-1 split of its four control computers occurred.
- Three seconds later, the split became 2-1-1.
- During troubleshooting, the remaining two computers disagreed creating a 1-1-1-1 split.
- But, none of the computers or their intercommunications were faulty!

What happened?

- The single fault was in a box (MDM FA2) that sends messages to the 4 computers via a multi-drop data bus that is similar to the MIL STD 1553 data bus.
  - This fault was a simple crack (fissure) through a diode in the data link interface.
The environment may create bugs

Byzantine fault
any fault that presents different symptoms to different observers
Byzantine failure
the loss of a system agreement service due to a Byzantine fault


Borrowed from Kevin Driscoll (Honeywell)

How about robotics?

https://www.youtube.com/watch?feature=player_detailpage&v=go0ZNQREXCQ#t=0