ICS 221: Software Analysis and Testing

Debra J. Richardson

Fall 2002
The Importance of “Testing”: the Ariane 501 explosion

On 4 June 1996, about 40 seconds after initiation of the flight sequence at an altitude of about 3700 m the launcher veered off its flight path, broke up and exploded.

The Inertial Reference System (IRS) computer, working on standby for guidance and attitude control, became inoperative. This was caused by an internal variable related to the horizontal velocity of the launcher exceeding a limit which existed in the software of this computer.

The backup IRS failed due to the same reason, so correct guidance and attitude information could no longer be obtained.

The limit was imposed according to the specification of the Ariane 4, when the software was ported to the Ariane 5 --whose flight specifications could exceed the imposed limit-- the specification was not changed and no test was performed using Ariane 5 actual trajectory data.
Why analysis and testing?

• Software is never correct no matter which developing technique is used
• Any software must be verified
• Software testing and analysis are
  – important to control the quality of the product (and of the process)
  – very (often too) expensive
  – difficult and stimulating
Testing vs. Analysis

• Testing, as dynamic analysis, examines individual program executions
  – Results apply only to the executions examined

• In contrast, static analysis examines the text of the artifact under analysis
  – Proof of correctness, deadlock detection, safety/liveness/other property checking, etc.
  – Results apply to all possible executions
Problems with “Static/Dynamic”

- Example: Model Checking
  - Evaluates individual executions/states
  - But is applied to all executions/states
- Example: Symbolic Execution
  - Examines source text
  - But summarizes individual executions/paths
- “Folding/Sampling” is a better discriminator

State-Space Exploration Pyramid

- Exhaustive Testing
- Infallible Prover
- Path Coverage
- Sampling
- Folding
- Threshold of Decidability
- Threshold of Tractability
- Effort
- Optimistic Inaccuracy
- Pessimistic Inaccuracy
- Mutation Testing
- Reachability Analysis
- Dataflow Analysis
- Complexity Measures
- Branch Coverage
- Statement Coverage
Software Qualities

**Dependability properties**
- correctness
- robustness
- safety
- reliability

**External properties (that can be verified)**
- timeliness
- interoperability

**External properties (that can be validated)**
- user-friendliness
- usability

**Process oriented (internal) properties**
- maintainability
- reusability
- modularity

**External properties (that can be verified)**
- maintainability
- reusability
- modularity
Validation vs. Verification

**Actual Requirements**

**Validation**
Includes usability testing, user feedback

**Verification**
Includes testing, inspections, static analysis

**Formal descriptions**

Verification or validation?
depends on the property

Example: elevator response

... if a user press a request button at floor i, an available elevator must arrive at floor i soon…

this property can be validated, but NOT verified
(SOON is a subjective quantity)

... if a user press a request button at floor i, an available elevator must arrive at floor i within 30 seconds…

this property can be verified
(30 seconds is a precise quantity)
You can’t always get what you want

Correctness properties are undecidable
the halting problem can be embedded in almost every property of interest
Getting what you need ...

We must make the problem of verification "easier" by permitting some kind of inaccuracy


"Easier" properties

Perfect verification

Optimistic inaccuracy (testing)

Pessimistic inaccuracy (analysis, proofs)

Sampling the input space

Folding the input space
The dimensions are not orthogonal

- Sufficient properties ~ pessimistic analysis
  - Analysis “false alarms” are in the area between desired property and checkable property
- Necessary properties ~ optimistic analysis
  - Faults go undetected if necessary conditions are satisfied
Impact of software on testing and analysis

- The type of software and its characteristics impact in different ways the testing and analysis activities:
  - different emphasis may be given to the same properties
  - different (new) properties may be required
  - different (new) testing and analysis techniques may be needed
Different emphasis on different properties

**Dependability requirements**
- they differ radically between
  - Safety-critical applications
    - flight control systems have strict safety requirements
    - telecommunication systems have strict robustness requirements
  - Mass-market products
    - dependability is less important than time to market
- can vary within the same class of products:
  - reliability and robustness are key issues for multi-user operating systems (e.g., UNIX) less important for single users operating systems (e.g., Windows or MacOS)
Different type of software may require different properties

- **Timing properties**
  - *deadline satisfaction* is a key issue for real time systems, but can be irrelevant for other systems
  - *performance* is important for many applications, but not the main issue for hard-real-time systems

- **Synchronization properties**
  - *absence of deadlock* is important for concurrent or distributed systems, not an issue for other systems

- **External properties**
  - *user friendliness* is an issue for GUI, irrelevant for embedded controllers
Different properties require different A&T techniques

- **Performance** can be analyzed using statistical techniques, but **deadline satisfaction** requires exact computation of execution times.
- **Reliability** can be checked with statistical based testing techniques, **correctness** can be checked with test selection criteria based on structural coverage (to reveal failures) or weakest precondition computation (to prove the absence of faults).
Different A&T for checking the same properties for different software

• Test selection criteria based on structural coverage are different for
  – procedural software (statement, branch, path,…)
  – object oriented software (coverage of combination of polymorphic calls and dynamic bindings,…)
  – concurrent software (coverage of concurrent execution sequences,…)
  – mobile software (?)

• Absence of deadlock can be statically checked on some systems, require the construction of the reachability space for other systems
Principles

Principles underlying effective software testing and analysis techniques include:

• **Sensitivity**: better to fail every time than sometimes
• **Redundancy**: making intentions explicit
• **Partitioning**: divide and conquer
• **Restriction**: making the problem easier
• **Feedback**: tuning the development process
Sensitivity:
better to fail every time than sometimes

• Consistency helps:
  – a test selection criterion works better if every selected test provides the same result, i.e., if the program fails with one of the selected tests, it fails with all of them (reliable criteria)
  – run time deadlock analysis works better if it is machine independent, i.e., if the program deadlocks when analyzed on one machine, it deadlocks on every machine
Redundancy: making intentions explicit

- Redundant checks can increase the capabilities of catching specific faults early or more efficiently.
  - Static type checking is redundant with respect to dynamic type checking, but it can reveal many type mismatches earlier and more efficiently.
  - Validation of requirements is redundant with respect to validation of final software, but can reveal errors earlier and more efficiently.
  - Testing and proof of properties are redundant, but are often used together to increase confidence.
Partitioned:
divide and conquer

- Hard testing and verification problems can be handled by suitably partitioning the input space:
  - both structural and functional test selection criteria identify suitable partitions of code or specifications (partitions drive the sampling of the input space)
  - verification techniques fold the input space according to specific characteristics, thus grouping homogeneous data together and determining partitions
Restriction: making the problem easier

- Suitable restrictions can reduce hard (unsolvable) problems to simpler (solvable) problems
  - A weaker spec may be easier to check: it is impossible (in general) to show that pointers are used correctly, but the simple Java requirement that pointers are initialized before use is simple to enforce.
  - A stronger spec may be easier to check: it is impossible (in general) to show that type errors do not occur at run-time in a dynamically typed language, but statically typed languages impose stronger restrictions that are easily checkable.
Feedback: tuning the process

• Learning from experience:
  – checklists are built on the basis of errors revealed in the past
  – error taxonomies can help in building better test selection criteria
Goals of Testing

• Find faults (“Debug” Testing):
  a test is successful if the program fails
• Provide confidence (Acceptance Testing)
  – of reliability
  – of (probable) correctness
  – of detection (therefore absence) of particular faults
Goals of Analysis

• Formal proof of software properties
  – restrict properties (“easier” properties) or programs (“structured” programs) to allow algorithmic proof
    • data flow analysis
    • necessary \( \exists \) sufficient conditions
  – compromise between
    • accuracy of the property
    • generality of the program to analyze
    • complexity of the analysis
    • accuracy of the result
Analysis and Testing are Creative

• Testing and analysis are important, difficult, and stimulating
  – Good testing requires as much skill and creativity as good design, because testing is design
• Testers should be chosen from the most talented employees
  – It is a competitive advantage to produce a high-quality product at acceptable, predictable cost
• Design the product and process for test
  – The process: for visibility, improvement
  – The product: for testability at every stage
Fundamental “Testing” Questions

• Test Adequacy Metrics: How much to test?
• Test Selection Criteria: What should we test?
• Test Oracles: Is the test correct?

How to make the most of limited resources?
Test Adequacy Metrics

• Theoretical notions of test adequacy are usually defined in terms of adequacy metrics/measures
  – Coverage metrics
  – Empirical assurance
  – Error seeding
  – Independent testing

• Adequacy criteria are evaluated with respect to a test suite and a program under test

• The program under test is viewed in isolation
Test Selection Criteria

• Testing must select a subset of test cases that are likely to reveal failures
• Test Criteria provide the guidelines, rules, strategy by which test cases are selected
  – requirements on test data -> conditions on test data -> actual test data
• Equivalence partitioning
  – a test of any value in a given class is equivalent to a test of any other value in that class
  – if a test case in a class reveals a failure, then any other test case in that class should reveal it
  – some approaches limit conclusions to some chosen class of faults and/or failures
Test Oracles/Correctness

- A test oracle is a mechanism for verifying the correct behavior of test execution
  - extremely costly and error prone to verify
  - oracle design is a critical part of test planning
- Sources of oracles
  - input/outcome oracle
  - tester decision
  - regression test suites
  - standardized test suites and oracles
  - gold or existing program
  - formal specification
Theory of Test Adequacy

Let

\[ P = \text{program under test} \]
\[ S = \text{specification of } P \]
\[ D = \text{input domain of } S \text{ and } P \]
\[ T = \text{subset of } D, \text{ used as test set for } P \]
\[ C = \text{test adequacy criterion} \]

- \( P \) is incorrect if it is *inconsistent* with \( S \) on some element of \( D \)
- \( T \) is unsuccessful if there exists an element of \( T \) on which \( P \) is *incorrect*
Subdomain-Based Test Adequacy

• A test adequacy criterion $C$ is subdomain-based if it induces one or more subsets, or subdomains, of the input domain $D$.

• A subdomain-based criterion $C$ typically does not partition $D$ (into a set of non-overlapping subdomains whose union is $D$).
An Example: Statement Coverage

Input Domain of System Under Test

System Under Test

Adequate Test Set
Test adequacy Axioms


• **Applicability**
  – For every $P$, there is a finite adequate $T$

• **Nonexhaustive applicability**
  – For at least one $P$, there is a non-exhaustive adequate $T$

• **Monotonicity**
  – If $T$ is adequate for $P$ and $T \subseteq T'$ then $T'$ is adequate for $P$

• **Inadequate Empty Set**
  – The empty set is not adequate for any $P$
Test Adequacy Axioms (cont.)

- **Anti-extensionality**
  - There are programs P1 and P2 such that P1 ° P2 and T is adequate for P1 but not P2

- **General Multiple Change**
  - There are programs P1 and P2 such that P2 can be transformed into P1 and T is adequate for P1 but not P2

- **Anti-decomposition**
  - There is program P with component Q such that T is adequate for P, but the subset of T that tests Q is not adequate for Q

- **Anti-composition**
  - There are programs P1 and P2 such that T is adequate for P1 and P1(T) is adequate for P2 but T is not adequate for P1;P2
Functional and Structural Test Selection Criteria

• Functional Testing
  – Test cases selected based on specification
  – Views program/component as black box

• Structural Testing
  – Test cases selected based on structure of code
  – Views program /component as white box
    (also called glass box testing)

Can do black-box testing of program by doing white-box testing of specification
Black Box vs. White Box Testing

**“BLACK BOX” TESTING**

**“WHITE BOX” TESTING**
Objective: Cover the software structure

Structural (White-Box) Test Criteria

- Criteria based on
  - control flow
  - data flow
  - expected faults
- Defined formally in terms of flow graphs
- Metric: percentage of coverage achieved
- Adequacy based on metric requirements for criteria
Flow Graphs

- Control Flow
  - The partial order of statement execution, as defined by the semantics of the language

- Data Flow
  - The flow of values from definitions of a variable to its uses
Subsumption and Covers

• C1 *subsumes* C2 if any C1-adequate T is also C2-adequate
  – But some T1 satisfying C1 may detect fewer faults than some T2 satisfying C2

• C1 *properly covers* C2 if each subdomain induced by C2 is a union of subdomains induced by C1
Structural Subsumption Hierarchy

What makes a program Testable?

• Testing assumes existence of a test oracle
• Program is non-testable if it requires extraordinary effort to determine test correctness
  – Often the case

• One solution: specification-based test oracles
  – Derive a test oracle from a specification
    • Possibly only for critical properties
    • Another argument for specification-based testing


Why Specification-based A&T?

• Specification states what system should do
  – this information should be used to drive testing
  – code-based testing detects errors of omission only by chance
  – specification-based testing is more likely to detect errors of omission

• Specifications enable formalized automation

• Specification-based analysis and testing should augment code-based testing


To detect, diagnose, and eliminate defects as efficiently and early as possible
Why Software Architecture-based A&T?

• SA-based A&T supports architecture-based and component-based software development

• Software Quality Assurance:
  – quality of the components and of their configurations
  – analysis in the context of connections and interactions between components

• Architecture Level of Abstraction:
  – components, connectors and configurations are better understood and intellectually tractable
  – analysis may address both behavioral and structural qualities, such as functional correctness, timing, performance, style, portability, maintainability, etc

Analysis at a higher level of abstraction makes the problem less complex
Specification-based Testing
Applied to Software Architecture

- During Requirements
  - specify critical system behaviors requiring highest assurance
- Architecture-based Testing
  - test structure for conformance to architectural design
  - test system and/or components against specified properties
- Component-based Testing
  - test components without knowing where they will be used
  - test component-based system consisting of OTS components
- Operational testing
  - monitoring of deployed software to perpetually verify behavior against residual test oracles and assumptions made during development-time analysis and testing
Argus-I: All-Seeing Architecture-based Analysis

- Iterative, evolvable analysis during architecture specification and implementation
- Specification-based architecture analysis
  - structural and behavioral analysis
  - component and architecture levels
  - static and dynamic techniques
- Current version
  - architectures in the C2-style (structure specification)
  - component behavior specification described by statecharts
- Future versions
  - generalize to other architectural styles and ADLs

Coordinated Architectural Design and Analysis with Argus-I

Architectural Element / Component Specification
- Create / Evolve
- Reuse / Import
- Analysis
  - Consistency Analysis
  - Reachability Analysis
  - Model Checking
  - Simulation
- Architectural Element / Component Specification
  - Argos/uml
  - ArchStudio

Component Implementation
- Develop / Deploy
- Analysis
  - State-Based Testing (DAS-BOOT)
- Component Usage Analysis (Retrospectors)

Architecture / Configuration Specification
- Create / Evolve
- Reuse / Import
- Argo/C2
- ArchStudio
- Analysis
  - Dependency Analysis
  - Interface Consistency
  - Model Checking
  - Simulation

Architecture Implementation
- Compose / Integrate
- Analysis
  - Monitoring & Debugging
  - Conformance Verification
  - Post-Deployment Assessment Monitoring (EDEM)

Data Integration (Knowledge Depot)
- Critiquing
- Component Implementation
Selected Current Work

- **Component-Based Dependence Analysis**
  - Static Analysis
  - Based on Architecture Specification and Component Implementation
  - For testing, maintenance and evolution


- **Software Architecture Monitoring**
  - Dynamic Analysis
  - Based on Architecture Specification and Implementation
  - For performance, conformance checking, understanding and visualization