Test development for distributed systems: Towards automation

Rachida Dssouli, Université de Montréal
Ferhat Khendek, Concordia University

May, 2000
Abstract

We give an introduction to methods and tools for testing communication protocols and distributed systems. In this context, we try to answer the following questions: Why are you testing? What are you testing? Against what are you testing?... We present the different steps of test automation and explain the industrial point of view (automatic test execution) and the research point of view (automatic test generation). The complete automation of the testing process requires the use of formal methods for providing a model of the required system behavior. We show the importance of modeling the aspects to be tested (the right model for the right problem!) and point out the different aspects (control, data, time and communication) that are of interest. We present the problem of testing based on existing models, such as finite state machines (FSMs), extended FSMs, timed FSMs and communicating FSMs, and give an overview of the proposed solutions and their limitations. Finally, we present our own experience in automatic test generation based on SDL specifications, and discuss some related work and existing tools.

NB- This course is based on previous tutorials done with Gregor von Bochmann
Table of content

• Introduction and motivation
• Software testing
• Testing sequential software
• Testing based on finite state models
• Testing complex systems
• Testing in a distributed environment
• The nature of test cases
• Towards the automation of testing activities
• Examples: Practical applications of automated testing
• Conclusions
Introduction and motivation

• Testing in the software development cycle
  – The software development cycle
  – Development of test cases (starting during analysis phase)
  – Analysis of test results, the oracle, diagnostics

• What results can we expected from testing?
  – Testing vs. verification
  – Finite test suite vs. infinite behavior
  – Definition of test suite
  – Conformance relations: What is a correct implementation?
  – The coverage problem and the fault models
  – Defining the correct behavior: modeling and specification languages
Why do we test?

• for detecting errors in the implementation /debugging

• for demonstrating conformance to a specification or users needs
  – e.g. protocol conformance testing

• for proving the correctness !!
Against what are testing?

Specifications:

• Users needs (Requirements)
• Objectives (specific)
• Informal specification
• Formal specification

The answer will help test team to establish a clear relationship between the system under test, the specification and the objective to satisfy.
Correctness and how to achieve it

How do we achieve the “correctness” of a given system? What is the impact of this process on the final software product?

- testing with partial coverage
- program proving (using theorem prover)
  - exhaustive Testing

The choice among these alternatives is based on:

- cost (function of # parameters: time, resources, human expertise,..)
- feasibility of proof or exhaustive testing
- the “target” quality
Limitations of testing approach

When to stop Testing?

May accept an infinite set of possible inputs when exhaustive testing (maximum power of error detection)

Huge Implementation Under Test

Question: How to choose a finite subset and preserve the maximum power of error detection?
Software Testing

• The process of systematically exercising software using a formal test plan for specific purpose of finding defect and errors.

• Testing is planned, using test case designs

• Test cases must be designed to have a maximum fault coverage, minimum time and effort.

“One cannot test a program to guarantee that is error-free and
a fundamental consideration in software testing is one of economics”

Art of Software Testing [G. J. Meyers]
Testing Taxonomy

• **Failure:**
  “system failures are system misbehaviors which have been detected”

• **Defect** “may or may not lead to a system failure, could be observable at a given level of abstraction (function)”

• **Fault** “is an execution of a given error, called error manifestation”

• **Error** “are made by human in design or implementations”
IEEE definitions

Relations
ERROR in design or implementation leads to DEFECT in Software

[Measuring Software Reliability, IEEE spectrum, August 1992, pp. 56-60]
Test cases design should start in early phase.

- Informal functional requirements
- Functional specification
- Detailed specification
- Implementation code
- Test cases
Software life cycle activities

(a) Creation of functional specification
(b) creation of detailed specification (design for testability)
(c) creation of implementation code
(d) validation of functional specification (e) validation of detailed specification, “
(f) validation of implementation code by informal walk-through
   and debugging tests
(h) design of test cases
(g) validation of implementation code by through formal testing procedure

Definitions:

Validation: “Are we building the right object?” (Latin validus - healthy, sound, effective)

Verification: “Are we building the object right?” (Latin veritas - truth or integrity)

“Bob Probert”
The purpose of test result analysis is to determine whether the trace of interactions observed during a test satisfies the requirements of the reference specification. This result, sometimes called verdict, is not easy to obtain in general; one sometimes refers to an “Oracle” to provide an answer. Clearly, such an oracle should be automated in order to make conformance testing manageable.”

Gregor von Bochamann
Oracle for deterministic specifications

Test suite

Apply

Specification

Apply

Black-Box Implementation

Expected output sequence

Compare the output Sequences

Observed output sequence

Non-conforming

Conforming

EQUAL?

No

Yes
Verification

An activity whose primary purpose and effect is to help demonstrate the integrity or self-consistency of the object being verified

- Detect Verification Errors (Defects)
  - Recall definition ("object was built right - Veritas")
  - Example 1: inconsistent requirements
  - Example 2: design defects found during ROOM/SDL walkthroughs (controlled simulations)
  - Example 3: code defects found in code inspection
  - Example 4: Poor code coverage in design test
  - Example 5: integration makes build "insane"

Dr. Robert L. Probert
Validation

- Detect validation errors (failures)
  - Example 1: requirements errors (omissions, invalid constraints, erroneous requirements)
  - Example 2: non-conformance of design to key client scenarios, missing high risk exception handling
  - Example 3: functional test violations, improper or inadequate exception handling during function/stress test
  - Example 4: regression test violations (at product test, system test and release process time)
Exception Handling

• What steps are, or should be, taken when a failure occurs?
  – Should priority be to keep system running, or to correct the problem immediately?
  – Trade-off: extensive checking in advance for exceptions, against system performance

• Effect of the failure: How does it propagate?
  – Loss of accuracy of output
  – function ceases
  – system ceases
  – network ceases
  – physical damage
  – life critical

Dr. Robert L. Probert
Software testing

State of the art
Sources of errors

- Incomplete and/or erroneous Specifications
- Improper Data Accessing
- Erroneous Logic
- Incorrect Calculations
- Code Errors (example: wrong constants)
Software Testing Methods

- **White-box testing/ structural testing** *(developers)*
  - Testing an entity with knowledge of its implementation, internal program structure, (called internal tests)
  - Performed early in the testing process.

- **Grey-box testing** *(design for testability, testers)*
  - Testing an entity with no knowledge of the code, with knowledge of the structure and may be the access/observation to additional interaction points.

- **Black-box testing / functional testing** *(testers)*
  - Testing an entity with no knowledge of its implementation,
  - Tests the product against requirements, functional specification,
  - Performed later in the testing process.
Types of testing

- **Unit Testing**: test of smallest entity (white box-testing)
- Integration Testing:
- Conformance testing: functional testing
- Interoperability testing
- Performance testing
- Robustness testing
- Configurations testing
- Security testing
- ...

```
Test Completeness (quality)

“Coverage is, in some sense, a test case value”

Coverage could be expressed in the following terms: (examples)

• Number of test cases passed
• Number of branches exercised
• Number of statements exercised
• Number of design components exercised
• Hours of execution
• Number of compile errors
• Percentage of Branches Exercised
• Percentage of Statements Exercised
• Percentage of Branches Exercised
• Percentage of Key Parameters Defined
• Percentage of Key Parameters Used
• Percentage of Conditions Covered
...
What are the aspects to test?

Testing all software aspects

- Data
- Control
- Time
Software faults

- **Sequencing**: e.g. missing alternative, improper nesting of loops, wrong logic expression in IF or WHILE statement
- **Data manipulation**: Arithmetic, overflow, etc. Calling wrong function, or using wrong operator. Wrong specification of data type or wrong format of data representation
- **Data flow**: Wrong initial values. Referencing wrong or non-initialized variable. Variable definition without usage
Black-box Techniques

Some techniques

• Model based techniques
• Class partitioning (Equivalence - Congruence)
• Value Selection strategies (internal/external boundaries),
• Cause-effect graph
• Syntactic testing
• Random testing
• Adaptive testing
• Transaction analysis
Structural Testing

• **Static Analysis**
  – Code inspection
  – Complexity estimation
  – Formal proof
  – Symbolic execution
  – Finite domain analysis

• **Dynamic Analysis**
  – Control graph based techniques
  – Mutation testing
  – Dynamic analysis of data flow
Testing Communication Protocols

Functional testing

The focus is Conformance testing
(ISO 9646 Conformance test methods)
Architecture of communication services and protocols

Abstract view

SAP

communication service
(e.g. Transport)

protocol entity

underlying service

Implementation view

local interface A

local interface B

Service view

communication service

Protocol view

impl.A

impl.B

underlying service
Protocol conformance testing

Objective: Interworking between different protocol implementations

Testing approaches:

- Interworking tests (problem: N x N possibilities)
- Conformance tests (if the protocol specification is well designed then any conforming implementation will interwork with any other conforming implementation)
- Conformance testing is black-box testing
Interoperability vs. conformance testing: an example

Protocol Standard
(e.g. TCP/IP)
Test Automation

- FSM extractor / by the user
- Definitions Files
- Formal Specification
- FSM Specification
- USER
- Abstract Test suite
- Completed Test Case TSL
- USER
- Validation (commercial tool)
- Validated Test Case
- Translation (SDL to C)
- Executable Test Case (C Code for example)
- TEST ENGINE
- TEST Results
- IUT

Definitions Files

Formal Specification

FSM extractor / by the user

Abstract Test suite

Completed Test Case TSL

Validation (commercial tool)

Validated Test Case

Translation (SDL to C)

Executable Test Case (C Code for example)

TEST ENGINE

TEST Results

IUT
Distributed architecture for protocol testing

• Independence between testers implies loss of coverage
• Queuing delays (loss of information about relative timings)
• Synchronization problem between testers
• External coordination between upper and lower testers
Distinguishing of the non-conforming implementations

Universe of all possible implementations of a given system

Question: How to choose a small (finite) test suite TS and obtain the maximum power of error detection?
Mutations -- Fault coverage

- Mutant is a change of the original (which is the specification, or a correct implementation)

- Fault coverage (always in respect to a given fault model) is
  - ratio between number of non-detected non-conforming implementations and total number of non-conforming implementations
Why fault models?

A fault model is a hypothetical model of what types of faults may occur in an implementation

- most fault models are “structural”, i.e. the model is a refinement of the specification formalism (or of an implementation model)
  - e.g. mutations of the specification or of a correct implementation
- it may be used to construct the fault domain used for defining what “complete test coverage” means
  - e.g. single fault hypothesis (or multiple faults)

A fault model is useful for the following problems:

- Test suite development for given coverage objective
- Formalization of "test purpose"
- For existing test suite: coverage evaluation and optimization
- Diagnostics
Behavior aspects of reactive systems
(to be specified -- to be tested)

(a) Temporal ordering of interactions
(b) Range of possible interaction parameters [not to be tested]
(c) Rules concerning actual values of parameters
(d) Coding of interactions (PDU's)

Corresponding specification languages:
(a) : FSMs, Petri nets, grammars, LTS
(b,c): Abstract data types
(a,b,c) : programming and specification languages
(b,d): ASN.1 (used for defining protocol messages)
Formal Description Techniques

• “FDTs” developed by ISO and ITU (CCITT) for OSI communication protocols and services
  – SDL (CCITT): version 1980: interconnected FSM's
    » version 1988: extended FSM, module interconnections
  – Estelle (ISO): extended FSM model, module interconnections
  – LOTOS (ISO): process algebra and abstract data types

• “Formal methods” (Europe)
  – Z (first order predicate calculus + sets)
  – VDM (similar concepts)

Comparison

– FDTs are executable
– LOTOS rendezvous more abstract -- SDL and Estelle use message passing paradigm, easier to implement
– graphic representation for SDL and LOTOS (optional)
Models of Specification and Implementation

- Conformance testing

![Diagram of Models of Specification and Implementation]

- abstract model of S
  - assumptions/test hypothesis
  - conformance relation
  - precise specification S
  - implementation I
  - abstract model of I
  - assumptions/test hypothesis
  - conformance relation
Specific Models for Specification

Use of Abstractions

« The right model for the right system! »

- Control aspect → FSM (Finite State Machine)
- Data aspects → high-level programming language
- Data and control aspects → Extended FSM (e.g. SDL)
- Communicating components → Communicating FSMs
  Comm. EFSMs (e.g. SDL)
Testing Sequential Software
Testing Sequential Software

- **White-box vs. black-box testing**
- **Coverage criteria**
  - Control flow criteria
  - Data flow criteria
  - Parameter variation
- **Test execution environments**
  - Test harness
  - Automatic test suite execution
- **Test result analysis**
  - Automatic oracle
Testing based on Finite State Models

• The finite state machine (FSM) model
• An infinite fault model
• Conformance relations: based on I/O sequences
• Testing based on FSM specifications
  – Fault model
  – Test derivation methods
    » Transition Tour
    » State identification methods
    » Fault coverage guarantees
    » Overview and assumptions
• Testing based on partially specified behavior
• Testing against non-deterministic specifications
  – Testing non-deterministic FSMs with input queuing
  – Test development based on specifications with rendezvous communication (e.g. LOTOS, LTS)
• Coverage analysis
An FSM Example

Mealy Machine

\[ M = < S, S_1, X, Y, D_s, \delta, \lambda > \]

- **S** \( \times \) **X**
- \( \delta : D_s \rightarrow S \)
- \( \lambda : D_s \rightarrow Y \)

- **S** = \{S_1, S_2, S_3, S_4\}
- X = \{1, 2\}
- Y = \{1, 2\}
- \( D_s = S \times X - \{<S_3, 1>\} \)

partially defined (specified), deterministic, initialized
Fault Model for Finite State Machine (FSM)

1) Output fault: point a in FSM fault model.

2) Transfer fault: point b in FSM fault model.

3) Transfer fault with additional states: point c in FSM fault model.

4) Additional or missing transitions: point d in FSM fault model.

5) additional or missing states
Example of a test suite

A test suite is a set of input sequences starting from the initial state of the machine.

\[ TS = \{ r.1.1.2.1, r.2.2.1.2.2 \} \]

<table>
<thead>
<tr>
<th>Test Case</th>
<th>MS</th>
<th>MI</th>
<th>M'I</th>
</tr>
</thead>
<tbody>
<tr>
<td>r.1.1.2.1</td>
<td>1.1.2.2</td>
<td>1.1.2.2</td>
<td>1.1.2.2</td>
</tr>
<tr>
<td>r.2.2.1.2.2</td>
<td>2.2.1.2.2</td>
<td>2.2.1.2.2</td>
<td>2.2.2.2.2</td>
</tr>
<tr>
<td></td>
<td>Conforming</td>
<td></td>
<td>Non-conforming</td>
</tr>
</tbody>
</table>

Pass TS  Fail to pass TS
An undetected faulty implementation

- Testing $\tilde{S}_{qe}$ is in general not solvable

$$TS = \{ r.1.1.2.1, r.2.2.1.2.2 \}$$

Expected output sequences: 1.1.2.2, 2.2.1.2.2

Consider input sequence: r.1.1.2.1.1
Expected output sequence: 1.1.2.2.2
Observed output sequence: 1.1.2.2.1
Conformance relations

“What properties must be satisfied by an implementation to conform to the specification?”

- **Deterministic specifications** (trace semantics): “to produce the same output as the specification, for any given input” (*equivalence relation*)
- **Partially defined behavior**: idem, but only for inputs for which the specification defines the behavior (*quasi-equivalence relation*)
- **Non-deterministic specifications**:
  - *Equivalence*: “all outputs that could be produced by the specification should also be producible by the implementation”
  - *Reduction of non-determinism*: “the output should be among those outputs that are possible according to the specification”; should the implementation be deterministic?
- **Failure semantics** (also considering possible blocking)
  - *reduction, “conformance”, testing equivalence*
- **Failure-trace semantics** (assuming testing can continue after the detection of blocking by offering other actions), etc.

Note: For non-deterministic implementations, the so-called *complete testing assumption* must be made: the IUT should exhibit during testing all its non-deterministic choices.
Possible changes made by a developer

- **Type 1**: change the tail state of a transition
- **Type 2**: change the output of a transition
- **Type 3**: add a transition; and
- **Type 4**: add an extra state.

No limitation on the number of such changes allows for an infinite set of possible implementations !!!
Fault model for FSM specifications

For the given transition:
• change the output (output fault)
• change the next state (transfer fault)
if a new state can be added, then
assume an upper bound on the
number of states in implementations.

For the example above, there are \((S \times O)^{S \times I} = 4 \times 7^4 \times 5 = 2820\) mutants
with up to 4 states. Among them, 36 mutants represent single
(output or transfer) faults, as only 9 transitions are specified.

An example of a very specific fault domain: Only the transitions
related to data transfer may be faulty. These are 4 transitions.
This results in “only” \(28^4\) mutants (faulty implementations in \(\Im_{mpl}\)).
Example implementations

\[ \text{not } \tilde{S}_q \]

\[ \text{S}_1 \]

\[ t_1:1/1 \]

\[ t_2:2/2 \]

\[ t_4:2/2 \]

\[ t_3:1/1 \]

\[ t_6:2/2 \]

\[ t_7:1/2 \]

\[ t_8:2/2 \]

\[ \text{M}_1 \]

\[ \text{M}_1' \]

\[ \text{S}_2 \]

\[ \text{S}_3 \]

\[ \text{S}_4 \]

\[ \text{M}_2 \]

\[ \tilde{S}_q \]

\[ t_5:1/2 \]

\[ t_6:2/2 \]

\[ t_7:1/2 \]

\[ t_8:2/2 \]

\[ t_3:1/1 \]

\[ t_4:2/2 \]

\[ t_2:2/2 \]

\[ 2/2.1/1.2/2 \]

\[ 2/2.1/1.2/1 \]
Test Derivation Methods
For a given FSM $S$, a transition tour is a sequence which takes the FSM $S$ from the initial state, traverses every transition at least once, and returns to the initial state.

Fault detection power

• Detects all output errors,
• There is no guarantee that all transfer errors can be detected.
The specification $S$

A transition tour is:
\[ a.a.a.b.b.b \]

The implementation $I_1$ contains an output error. Our transition tour will detect it.

The implementation $I_2$ contains a transfer error. Our transition tour will not detect it.
An input sequence is a distinguishing sequence \((DS)\) for an FSM \(S\), if the output produced by the FSM \(S\) is different when the input sequence is applied to each different state. A \(DS\) is used as a state identification sequence.

- Detects all output errors,
- Detects all transfer errors,
- A \(DS\) may not be found for a given FSM.
The specification S

A distinguishing sequence is:
\[ b.b \]

If we apply it from:
- state 1 we obtain \( y.y \)
- state 2 we obtain \( y.x \)
- state 3 we obtain \( x.y \)

A test case which allow the detection of the transfer error is:
\[ a.b.b.b \]

If we apply it from the initial state of:
- the specification we obtain \( x.x.y.y \)
- the implementation we obtain \( x.x.x.x \)
The test cases are:

state 1: \(a.b.b\)  
\(b.b.b\)

state 3: \(a.a.b.b\)  
\(a.b.b.b\)

state 2: \(b.a.b.b\)  
\(b.b.b.b\)

Test case structure:

\textit{preamble} .tested transition .state identification
The *UIO*-method can be applied if for each state of the specification, there is an input sequence such that the output produced by the machine, when it is initially in the given state, is different than that of all other states.

The *UIO*$_v$-method is a variant of the *UIO*-method. it check the uniqueness of the applied identification sequences on the implementation, meaning that each identification sequence must be applied on each state of the implementation and the outputs are compared with those expected from the specification.

*UIO*-Method [Sabnani 88] and *UIO*$_v$-Method [Vuong 89]
The specification $S$

UIO sequences are:
- state 1: $a.b$
- state 2: $a.a$
- state 3: $a$

A transition cover set is:
$P = \{e, a, a.b, a.a, b, b.a, b.b\}$

The test sequences generated by the UIO-method are:
$r.a.b, r.a.a, r.a.b.a.b, r.a.a.a.a,$
$r.b.a.a, r.b.a.a.b, r.b.b.a$
Method $W$ [Chow 78]

The $W$-method involves two sets of input sequences:
- $W$-set is a characteristic set of the minimal FSM, and consists of input sequences that can distinguish between the behaviors of every pair of states.
- $P$-set is a set of input sequences such that for each transition from state $A$ to state $B$ on input $x$, there are input sequences $p$ and $p.x$ in $P$ such that $p$ takes the FSM from the initial state into state $A$. 
The specification $S$

We assume the existence of a reset transition with no output (r/-) leading to the initial state for every state of $S$

A characterization set is $W=\{a, b\}$
- for state 1: $a/e, b/f$
- for state 2: $a/f, b/f$
- for state 3: $a/f, b/e$

A transition cover set for the specification $S$ is:
$P=\{e, a, b, b.c, b.a, b.b, c, c.a, c.c, c.b\}$

The W-method generates the following test sequences:
$r.a, r.b, r.b.a, r.b.b, r.c.a, r.c.b, r.a.a, r.a.b, r.b.c.a, r.b.c.b, r.b.a.a, r.b.b.b, r.c.a.b, r.c.c.a, r.c.c.b, r.c.b.a, r.b.a.b, r.b.b.a, r.c.a.a, r.c.b.b$
This method is a generalization of the $\text{UIO}_v$ method which is always applicable. It is as the same time an optimization of the W-method. The main advantage of the $W_p$-method, over the W-method, is to reduce the length of the test suite. Instead of using the set $W$ to check each reached state $s_i$, only a subset of $W$ is used in certain cases. This subset $W_i$ depends on the reached state $s_i$, and is called an identification set for the state $s_i$. 

**Wp method [Fujiwara 90]**
The specification $S$

- For state 1: $a/e$
- For state 2: $a/f$, $b/f$
- For state 3: $b/e$

We assume the existence of a reset transition with no output ($r/-$) leading to the initial state for every state of $S$.

The identification sets are:
- $W_1 = \{a\}$, distinguishes the state 1 from all other states
- $W_2 = \{a, b\}$, distinguishes the state 2 from all other states
- $W_3 = \{b\}$, distinguishes the state 3 from all other states

$W = \{a, b\}$
- For state 1: $a/e$
- For state 2: $a/f$, $b/f$
- For state 3: $b/e$

Example of $W_p$ method (1/3)
A state cover set for the specification $S$ is: $Q=\{e, b, c\}$

A transition cover set for the specification $S$ is:
$P=\{e, a, b, b.c, b.a, b.b, c, c.a, c.c, c.b\}$

Based on these sets, the Wp-method yields the following test sequences:
• Phase 1: the test sequences for this phase are
  $Q.W.=\{r.a, r.b, r.b.a, r.b.b, r.c.a, r.c.b\}$
• Phase 2: the test sequences for this phase are
  $(P-Q).W_i=\{r.a.a, r.a.b, r.b.c.a, r.b.c.b, r.b.a.a, r.b.b.b, r.c.a.b, r.c.c.a, r.c.c.b, r.c.b.a\}$
Example of Wp method (3/3)

A faulty implementation $I$

$I$ contains a transfer error $2\rightarrow a/f\rightarrow 1$ (fat arrow) instead of $2\rightarrow a/f\rightarrow 2$ as defined in the specification $S$

- The application of the test sequences obtained in Phase 2 leads to the following sequences of outputs:
  
e.f, e.f, f.f.f, f.f.f, f.f.e, f.f.e, e.f.f, e.e.f, e.e.f, e.e.e

- The output printed in bigger size is different from the one expected according to the specification. Therefore, the transfer error in the implementation is detected by this test sequence.
Test derivation based on FSM (Resumé)

- **Transition tour**
  - guaranteed coverage only for output faults

- **Methods using state identification**
  - with coverage guarantee for output and transfer faults. Three cases:
    » number of states same for implementation I and specification S
    » number of states for I possibly larger than for S, but **bounded**
    » coverage only for a selected set of transitions (“fault function”)

- **Methods without coverage guarantee**
  - Hand made test suite without test derivation procedure

- **Single long test sequence vs. set of shorter test cases** (e.g. test case for specific transition, “test purpose”)

- **Usually, each test case requires reset to initial state; “correct reset assumption”**
Example of implementation with additional state

Specification

Impl. 1

Impl. 2
Transition Tour example

Test hypothesis: Initially connected machine

Transition tour
TT: $t_1, t_4, t_9, t_2, t_3, t_6, t_7, t_8$
TT (input/expected output): a/1.b/2.a/1.a/2.b/2.a/1.b/2.a/2.b/2
All state identification Methods

Distinguishing Sequence, UIO, W

**Test hypothesis**
- H1) Strongly connected machine
- H2) Contain no equivalent states
- H3) deterministic
- H4) Completely specified machine
- H5) the failure which increases the number of states doesn’t occur

The method is applied in two phases from the initial state
- phase 1) $\alpha$-sequence to check that each state defined by the specification also exist in the implementation.
- phase 2) $\beta$-sequence to check all the individual transitions in the specification for correct output and transfer in the implementation.
Assume that Reset transition r/- exist

Q1) Verify if a.a is a DS for S and explain why?
Q2) Find a DS different from “a.a” with length 2 for S
Assume the reset is exist and bring the machine from any state to the initial state.

a) Find characterization set $W$ and generate the set of test cases for the specification $S$ using the $W$ method.
b) Does $S$ have a DS sequence? If not explain why?
W method

\[ W = \{a, b\} \]

\[ S_0 : a/0, b/1 \]
\[ S_1 : a/1, b/0 \]
\[ S_2 : a/0, b/0 \]

\[ Q = \{a, b\} \]
\[ P = \{a, b, a.b, a.a, b.a, b.b\} \]

Phase 1, \( Q.W = \{r.a, r.b, r.a.a, r.a.b, r.b.a, r.b.b\} \)

Phase 2, \( (P-Q).W = \{r.a.b.a, r.a.b.b, r.a.a.a, r.a.a.b, r.b.a.a, r.b.b.a, r.b.b.b\} \)
Examples Suite

Derive a DS of length up to 2 for S
a.b is a DS for S

<table>
<thead>
<tr>
<th>State</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>a.b</td>
<td>a.b</td>
<td>a.b</td>
<td>a.b</td>
<td>a.b</td>
<td>a.b</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>1.0</td>
<td>0.0</td>
<td>0.1</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Comment: “a” as input at each state will loop on the state, sequence of “a.a.” cannot be a DS, the output will be 0.0.. or 1.1..
Q set: permits to reach each state from the initial state

\[ Q = \{ , b,b,b\} \]

The first “b” to reach the state \( S_2 \)

“b.b” to reach the state \( S_1 \).

\( P \) set is transition cover, permits to execute each transition at least one starting from the initial state

\[ P = \{ , a, b, b.a, b.b, b.b.a, b.b.b\} \]

more than one \( P \) set may exist, this depends on the alternative paths that the automata may have.
The goal of the Phase 1 is identification of the states in the implementation

\[ DS = a.b, \quad Q = \{ e, b, b.b \}, \quad P = \{ e, a, b, a.b, b.b, b.b.a, b.b.b \} \]

Phase 1

\[ Q.DS = \{ r.a.b, r.b.a.b, r.b.b.a.b \} \]

Expected output of phase 1 is:

\[ \{-0.1, -1.0.0, -1.0.1.0\} \]

Phase 2 (DS in bold)

\[ P.DS = \{ r.a.b, r.a.a.b, r.b.a.b, r.b.a.a.b, r.b.b.a.b, r.b.b.a.a.b, r.b.b.b.a.b \} \]

\[ \{-0.1, -0.0.1, -1.0.0.0, -1.0.1.0, -1.0.1.1.0, -1.0.0.0.1\} \]

Note that, the test suites for phase 1 and 2 should be Derived from the specification and applied to the implementation to check it for output and transfer faults.
Apply the transition tour to the implementation I and comment

The implementation I has a transfer fault, the fault is not detected by Transition tour.
Transition tour detects all output faults but Doesn’t guarantee the detection of transfer faults
The goal of the Phase 1 is identification of the states in the implementation

\[ DS = a.b, \quad Q = \{ a, b, b.a, b.b, b.b.a, b.b.b \}, \quad P = \{ a, b, b.a, b.b, b.b.a, b.b.b \} \]

Phase 1

\[ Q.DS = \{ r.a.b, r.b.a.b, r.b.b.a.b \} \]

Expected output of phase 1 is:

\[ \{-0.1, -1.0.0, -1.0.1.0\} \]

\[ \{-0.1, -1.0.0, -1.0.1.0\} \]

observed outputs from I

Phase 2 (DS in bold)

\[ P.DS = \{ r.a.b, r.a.b, r.b.a.b, r.b.b.a.b, r.b.b.a.b, r.b.b.b.a.b \} \]

Expected output

\[ \{-0.1, -0.0.1, -1.0.0.0, -1.0.1.0, -1.0.1.1.0, -1.0.0.0.1\} \]

observed output from I, transfer fault detected
Specification S

Derive a UIO sequence for S

State | S0 | S1 | S2 | S0 | S1 | S2 | S0 | S1 | S2
-----|----|----|----|----|----|----|----|----|----
Input | a  | a  | a  | b  | b  | b  | c  | c  | c  | a.c | a.c | a.c
Output| 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0.1 | 0.0 | 1.1

UIO state S0 = c/1
UIO state S2 = a/1
UIO state S1 = a/0.c/0
Testing Assumptions and Hypothesis

Objective to Reduce the Set of Test Cases
Assumptions about specifications

- **completeness**: completely specified or partially specified
- **connectedness**: strongly connected or initially connected
- **reducibility**: reduced or non-reduced
- **determinism**: deterministic or non-deterministic
Assumptions about implementations

- **Deterministic**
- **Completely defined**
  - react to any input
- **Limited extra states**
- **Reliable reset**
  - not necessary
Regularity, a testing assumption

- This type of assumption allows to limit testing to a finite set of behaviors in the case of systems that exhibit an infinite behaviors. Examples are
  - programs (or specifications) with loops and integer input and output parameters
  - finite state machines
  - reactive systems, en general

- Principle: assume that the implementation has a “regular” behavior, which means that the number of control states of the implementation is limited.

  - If the number of states is not bigger than the corresponding number of states of the specification, then all loops (of the specification) have to be tested only once.
    » This is the idea behind the FSM fault model where the number of implementation states is limited to n, or to some number m > n.
    » This is also the idea behind certain approaches for testing program loops and for testing in respect to specifications in the form of abstract data types.
Independency, a testing assumption

• **Principle:**
  - The different submodules of the system under test are independent, and faults in one module do not affect the possibility of detecting the faults in the other modules.

• **This is a controversial assumption:**
  In most complex systems, modules or components are dependent. The reasons are:
  » they share resources (e.g. memory)
  » they have explicit interactions

• **Example:**
  - several connections supported by a protocol entity
    » test only one connection in detail (it is independent of the others)
    » the others need not be tested, since they are all equal (uniformity assumption, see below)
The independency relation is a reasonable assumption in certain cases.

Example:

![Diagram showing the concept of independent entities](image-url)
Uniformity, a testing assumption

• Uniformity assumption / Congruence
  – Origin: Partition Testing [Weyuker 91]

• Principle
  “There exist similar behaviors. If they are grouped under an equivalence relation, then it is sufficient to test one behavior of each equivalence class for conformance testing.”

• Special cases:
  – Principle of partition testing: Apply test for at least one representative for each partition of the input domain (software testing, EFSM testing)
  – Equivalent actions for EFSM
  – Equivalent states for FSM
Fairness in respect to non-determinism

• Many systems have a non-deterministic nature. In particular, the parallelism of distributed systems introduces many possible interleaving of individual actions within the different system components.

• The assumption is that all the execution paths effectively realized during testing cover all paths that are pertinent for detecting the possible implementation faults.
Partially defined FSM’s

Non-specified transitions need not be tested. However different interpretations of “undefinedness” have an impact on testing:

• **completeness assumption**
  – non-specified transition is implicitly defined, e.g. stay in same state (as in SDL), or go to an error state
  – methods for completely defined FSM’s may be applied, however, test will rely on implied transitions

• “**don’t care**”
  – no specific behavior is specified
  – non-specified transitions must be avoided by test cases
  – robustness tests may be applied to check the reaction of the implementation for non-specified situations

• “**forbidden**”
  – not possible to invoke non-specified transitions
Fault Coverage Evaluation
Methods for Fault Coverage Evaluation

The definition of fault coverage always depends on fault model!

- **Exhaustive mutation analysis**
- **Monte-Carlo simulation method**
- **Deciding completeness**
  - minimize an FSM which is given in the form of the TS, if its minimal form is equivalent to the given FSM then TS is complete (the max # states is assumed), otherwise it is not complete [see Yao]
- **Structural Analysis**
  - it evaluates the fault coverage of a given test suite by directly analyzing the test suite against the given FSM. Count the number of states distinguished and transitions checked by the test suite. A numeric measure easy to evaluate (linear complexity) [see Yao]
- **Different possible measures**
  - compare number of implementations (common approach)
  - compare the log of number of implementations (corresponds to counting “transitions covered”) [called “order coverage” by Yao]
Testing Complex Systems
Testing complex systems

• Introductory comments
• Test development for extended FSM specifications (e.g. SDL)
  – specification paradigm
  – approaches to test development
• Testing systems consisting of several modules
• Testing real-time properties
• Test hypothesis and assumptions
• Comments
  » Comments on assumptions
  » Explicit test purposes vs. optimizations
  » Cost models for the testing process
  » A more global cost model concerning testing
Example of extended FSM specification

INRES
Responder

Input parameter of DT interaction: $N$ -- # of data sent by the initiator

Local variable: $NE$ -- # of data expected by the responder.

Predicates of transitions: $DT(N)\&N=NE$; $DT(N)\&N\neq NE$.

Actions associated with transitions: $NE:=NE\&1$; $NE:=1$. 
Testing in respect to EFSM specifications: Issues

- Executability of execution paths including unlimited loops is undecidable
- State identification (including values of local variables) in the case of black-box testing
- Data flow analysis and coverage of data flow faults *
- Systematic parameter variation for testing *

* like in software testing

Much of this is similar to the issues in software testing, however, for EFSM’s one considers sequences of inputs and outputs (not a single function).
Approaches to testing EFSMs

• **Separation of control flow (FSM) and data flow:** using FSM testing techniques for the former and dataflow-related techniques for the latter

• **Normal form EFSM specifications:** putting all control flow aspects into the FSM part (e.g. replacing a transitions including an *if* statement in the action by two simple transitions with predicates, one for TRUE, one for FALSE)

• **Restricting the form of predicates and actions to make most issues decidable,** e.g. Pressburger arithmetic +, -, *, =, <, >

• **Unfolding:** transform into FSM model by limiting the range of the types of interaction parameters and variables (see INRES example)
Example of unfolding

**INRES**

**Responder**

**Input parameter of DT interaction:** N -- # of data sent by the initiator

**Local variable:** NE -- # of data expected by the responder.

**Predicates of transitions:** DT(N)&N=NE; DT(N)&N≠NE.

**Actions associated with transitions:** NE:=NE&1; NE:=1.

The above EFSM is equivalent to the following FSM

Note: “Unfolding” of input parameter and variables is possible provided that their type is restraint to small domains.
Testing of timed specifications

• Need for a timed specification model
• Need for a timed conformance relation

In our work, we use as specification model an FSMs augmented with timers and counters

Example:

\[ \begin{align*}
    &i1/o, \text{start } T, C=0 \\
    &i2/o \\
    &i3/o \\
    &T&C<\text{max/o1, start } T, C=C+1 \\
    &T&C=\text{max/o2} \\
    &i1/o, \text{stop } T \\
    &i2/o, \text{stop } T
\end{align*} \]
Testing a fault model for timers and counters

FAULT
• start is missing
• stop is missing
• T is shorter
• max is less
• reached state is wrong
• unexpected start
• unexpected stop

TEST
• wait for expiration
• wait if T expires
• measure time
• wait max times
• apply state identifier
• wait if T expires
• wait if T expires

Some assumptions:
– timers and counters are fail-safe
– time-out causes a visible output
– tester measures time intervals between events
Example of a timed test case

An example:
To test whether \textit{start} on \textit{i1} is implemented

We construct the test case:
\begin{itemize}
\item a preamble to the starting state of the transition
\item input \textit{i1}
\item wait a certain time
\item check the output, if any
\item check the final state of the transition by applying a state identifier
\end{itemize}
Testing in a distributed environment

- Test architectures for OSI protocol conformance testing
- Problems of distributed testing
  - observability
  - controlability
  - synchronization of different testers
- Testing embedded components
- Test management in a distributed environment
A general distributed test architecture

Issues in testing multiple ports architecture. How to generate test suite for each tester from a global specification? How to synchronize different testers?
Local and distributed architectures

**Distributed architecture**

Difficulties:

- Independence between testers implies loss of coverage
- Queuing delays (loss of information about relative timings)
- Synchronization problem between testers
- External coordination between upper and lower testers

---

Local architecture
All interactions directly observable and controllable
Embedded protocol testing

Example architecture:
Embedded test architectures

Embedded testing (ongoing research):

- Certain faults can be masked and cannot be easily activated: How can one formally characterize detectable and undetectable faults in an embedded IUT?

- Given an IUT embedded within a system, the ability to control its inputs and observe its outputs is decreased: How can one transform the specification FSM into a specification that correspond to what is visible from the external point of view?

Note: For a deterministic FSM specification, its visible behavior may correspond to a non-deterministic FSM.
The nature of test cases

• The structure of test cases
  – Preamble, test body, postamble

• What properties of test cases must be specified?
  » Different levels of abstraction
  » Abstract test cases
  » The real interface used for testing
  – Example: Tests for the ATM signaling protocol

• TTCN: a language for defining protocol conformance test cases
Test script languages ISO 9646-3

In which language should the test cases be defined?

– Programming language, e.g. C
  » same as implementation language, ease of use for in-house testing
  » low level of abstraction, often leads to lengthy programs
  » implementation-dependent, not so interesting for standardized conformance tests

– TTCN
  » standardized by ISO and IUT-T (CCITT) for describing protocol conformance tests
  » difficult to understand by people not familiar with language
  » need for editing and translation tools

– SDL
  » standardized by IUT-T
  » has an intuitive graphical representation for many aspects
  » also used for protocol specifications
  » need for editing and translation tools

Our choice: SDL
Towards the automation of testing activities

- Possibilities for partially automating test suite development
  - Tools for test suite development from state transition models
  - Tools for test suite development from SDL or Estelle specifications
  - TTCN support tools

- Test execution

- Possibilities for partially automating result analysis and diagnostics
  - Obtaining automatically an oracle for a given specification
  - Automatic FSM diagnostics

- Automatic coverage analysis
Data Flow Analysis Techniques

An example of static analysis techniques
Data Flow Analysis (def/use)

Def Use technique: establish relations between variable definition and its uses.

An example:
read (x) definition of variable x,
write (x) output usage of x (referenced),
if (x=1) then x := 7 predicate usage of x, definition of x,
a[i] := x definition of a[i], usage of i and x,
x := x+1 calculus usage (right side) and definition of x (left side)

From S. Xanthakis, M. Maurice, A. de Amescua, O. Houri, L. Griffet
Test & Contrôle des logicielles - méthodes, techniques et outils
“Fault” detection with Def Use

Program p(input, output)

Var x, y, z, a, b, c : integer

begin
read (c);
x := 7;
y := x + a;
b := x + y + a;
c := y + 2*x + z;

Variable Def/Use chain technique

<table>
<thead>
<tr>
<th>Variable</th>
<th>du-chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>duuuu</td>
</tr>
<tr>
<td>y</td>
<td>duu</td>
</tr>
<tr>
<td>z</td>
<td>u</td>
</tr>
<tr>
<td>a</td>
<td>uu</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td>ddu</td>
</tr>
</tbody>
</table>

p (c) = ddu

- Usage without definition (a, z)
- Sequence of definitions without usage between (c)

General static analysis based on Def/usage chain

1) u... variable initial value is missing
2) ...dd.. consecutive definitions, the first is useless
3) ...d last definition is useless
Exercise -1

Find the def/use chain and faults of the following program.

```plaintext
a := b
if (a = 1) then a := 6
else a := b;
if (x = 1) then b := a
else a := b + c + 2
```
Exercise 1 - Results

\[ \pi (a) = dudu \quad \text{or} \quad dudd \]

\[ \pi (b) = ud \quad \text{or} \quad uu \quad \text{or} \quad uud \quad \text{or} \quad uu \]

\[ \pi (x) = u \]

\[ \pi (c) = u \quad \text{or} \quad 1 \quad (1 \text{ means no definition no use}) \]
Find the def/use chain and faults.

\[\text{sum} := 0\]
\[i := 1\]
\[\text{while } (i \not\geq N) \text{ do} \]
\[\quad \text{begin}\]
\[\quad \text{sum} := \text{sum} + 1;\]
\[\quad i := i + 1;\]
\[\quad \text{end}\]
Exercise 2 - Results

\[ \pi \text{ (sum)} = d /or/ dud /or/ dudud, ... \]
\[ \pi \text{ (i)} = du /or/ duuudu /or/ duuuduuudu, .... \]
\[ \pi \text{ (N)} = u \]
Symbolic execution

\[ a := a \times a \]
\[ x := a + b \]
\[ \text{if } x = 0 \text{ then } x := 0 \]
\[ \text{else } x := 1; \]

- Distinguish between variable value and symbolic value
Symbolic execution

\[ a = A, \ b = B, \ x = X \]

\[ a = A \cdot A, \ b = B, \ x = A \cdot A + B \]

\[ a = A \cdot A, \ b = B, \ x = A \cdot A + B \]

\[ A \cdot A + B = 0 \]

\[ A \cdot A + B \neq 0 \]

\[ a = A \cdot A \]
\[ b = B \]
\[ x = 0 \]

\[ a = A \cdot A \]
\[ b = B \]
\[ x = 1 \]
Finite Domain Analysis Technique
Finite Domain Analysis

To illustrate finite domain analysis, we will use the following program:

```pascal
read (x,y);
if (y>= x ) then 
    begin 
        if (y >5) then d :=1 
            else c :=2;
    end 
if (x + y <4) then c := c + 10 
else  c := c + 20;
writeln (c);
```

```
<table>
<thead>
<tr>
<th>D1</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>TFF</td>
</tr>
<tr>
<td>D3</td>
<td>TFT</td>
</tr>
</tbody>
</table>
```

```plaintext
| FT |
```
For inputs x and y between 0 and 10 and an output c. We have 3 conditions:

C1  y >= x
C2  y > 5
C3  x + y < 4

D1 : y = x, D2 : y = 5 and D3: y = 4 - x

TFF means that
the first condition is True
the second condition is False
and the third condition is False

Remarks:

We deduce from this geometric representation, that FFT and TTT doesn’t exist.

Why?

FFT : If C1 is false we deduce that C2 cannot be reached (case of nested conditions)
TTT: D2 domain and D3 domain do not intersect, C2 and C3 can not be satisfied simultaneously
Control Flow Techniques

An example of Dynamic analysis technique
Dynamic Analysis Technique

What is a control flow graph?

Program P

if $x \leq 0$ then $x := -x$
else $x := 1 - x$
if $x = -1$ then $x := 1$
else $x := x + 1$
writeln(x)
What are paths that I can exercise?

4 control paths from $G$

$\beta_1 = [a, b, d, f, g]$
$\beta_2 = [a, b, d, e, g]$
$\beta_3 = [a, c, d, f, g]$
$\beta_4 = [a, c, d, e, g]$

$M$ is set of paths, it could be written in the following expressions:

$M = \beta_1$ or $\beta_2$ or $\beta_3$ or $\beta_4$

$M = \beta_1 + \beta_2 + \beta_3 + \beta_4 = adfg + abdeg + acdfg + abdeg$

$M = a(bdf + bde + cdf + cde)g = a(b + c)d(e + f)g$
Test Executability Problem

Find x values that excites each path.

Executable paths:

{ x = 0} excites the path $\beta_2 = [a, b, d, e, g]$

{ x = 2} excites the path $\beta_3 = [a, c, d, f, g]$

{ x = 1} excites the path $\beta_4 = [a, c, d, e, g]$

Non executable path:

$\beta_1 = [a, b, d, f, g]$ ????
Path Criteria

Weyuker & Rapps 1985

Graph coverage criteria

- All- paths
- All- du- paths
- All- uses
- All c-uses/
  Some p-uses
- All defs
- All p-uses/
  Some c-uses
- All edges
- All nodes

This family of criteria is partially ordered by strict inclusion. The test suite size is in relation with the criteria to satisfy.
Fault Coverage

“A successful test case detects the presence of defect by manifesting an observable behavior”

Mayers
Test Completeness (quality)

“Coverage is, in some sense, a test case value”

Coverage could be expressed in the following terms: (examples)

- Number of test cases passed
- Number of branches exercised
- Number of statements exercised
- Number of design components exercised
- Hours of execution
- Number of compile errors
- Percentage of Branches Exercised
- Percentage of Statements Exercised
- Percentage of Branches Exercised
- Percentage of Key Parameters Defined
- Percentage of Key Parameters Used
- Percentage of Conditions Covered
...
Commercial Tools

• Telelogic Tools
  – SDT
  – TAU
  – ObjectGeode (from Verilog, now is part of Telelogic products)

• Cinderella
Conclusions (I): Protocol testing vs. general software testing

The following characteristics distinguish protocol testing methods from those developed for software in general:

- Black-box approach with well-defined (but not formal) specification
- Typical testing interfaces
  - at the specification level: (1) queues (FSM, SDL), or (2) rendezvous (LTS, LOTOS) [not operation call, as in software engineering]
  - at the implementation level: operation calls (APIs)
- Specifications may be non-deterministic, a variety of conformance relations have been considered
- Multiple PCO’s and various, possibly non-local, test architectures
Conclusions (II):
Application of protocol testing methods in software testing

The above characteristics make the protocol testing methods applicable to software testing, specifically for reactive systems.

( in addition to the application area of communication software, also to such applications as process control, real-time systems, distributed systems management, etc. )
Principles

• Software product/system releases or updates will contain defects and *will fail* in the field

• Design, verify (syntactic) and validate (semantic) accordingly

• Code, inspect and design test accordingly

• Integrate and test accordingly

• Product test, performance test, conformance test, regression test accordingly

• A successful test is one which detects the presence of a (serious) error by bringing about an (observable) system failure.

Dr. Robert L. Probert
A Strategy:

I) find error early
II) detect the presence of most high-cost errors
III) maximize return on test investment

Dr. Robert L. Probert
Recommendations

• **Personal Software Process** (Plan/Measure/Assess/Improve)

• **Requirements** Capture & Validation (Black-box)
  – Use cases - emphasize boundaries and exception cases
  – risk-directed scenarios (time-sequence diagrams, MSCs) with error-guessing
  – Causes-effect graphs (extended to real-time systems)
  – (E)FSM modeling with verification and validation

Dr. Robert L. Probert
Recommendations (part 2)

• Systematic DESIGN Verification (Grey-box)
  – Structured design
  – CASE tool support + baseline inspection coverage criteria

• CODE-base Verification (White-box Analysis)
  – Static Verification
    » code inspections using coverage analysis
  – Dynamic Verification and Validation
    » Decision/Condition Code Coverage
    » Simple Input-def to output-use Path Coverage
    » Risk based Exceptions Functional Testing

Dr. Robert L. Probert
Recommendations (part 3)

• Integration testing
  – Incremental sanity assurance
  – All Pairwise collaborations tests

• Product Assurances
  – Robustness Testing
    » Focus on exceptions & HMI
  – Configuration Verification & Capabilities Validation
  – Automated Conformance Verification
  – Automated Performance Assessment
Recommendations (part 4)

- **System Assurances/Acceptance:**
  - Automated Regression Testing
    » No Existing Customer Requirements Violated
    » No existing Customer Functionality Broken
    » Backward Compatibility Tested
  - Automated Reliability Testing
    » Under Intensive in-house “soak” Testing

**Code is stable + Failure Intensity Objective is Met**

Dr. Robert L. Probert
Conclusions/Recommendations

• Best people
• Best management support
• Adequate resources
• Client-centered development process
• Most Appropriate Quality Metrics
• Best Tool Support
• Most effective Investment of Resources
• Most Useful Methods

Dr. Robert L. Probert
Organizational strategies

Industrial Sector

Invest in education, training, acquiring, working environment and management support, and professional growth/updates of the highly qualified human resource.

Educational/Public Sector

Focus on foundations - discrete mathematics, pragmatic statistics, automation, industrial-strength state machines, logic, international standards, software quality engineering.

Broaden to ethics and technology, business sense, societal impact and responsibility, privacy and human rights, safety and liability.

Academic/Research Sector

Invest in education and research into models and automated methods which leverage human intelligence, domain expertise, industrial scale, cost, time, risk.