Formal Methods in Testing Software Architectures

Antonia Bertolino
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
ISTI-CNR, Pisa

Paola Inverardi, Henry Muccini
Dipartimento di Informatica, Università dell'Aquila

Talk ingredients

Formal Methods
Testing
Software Architectures

Formal Methods
in Testing
Software Architectures

Talk ingredients

Testing

☺ testing expertise

- Improving and automating structural testing
- Precise dataflow-based test criteria (to prevent untestedness)
- Operational Testing (for reliability evaluation)
- Testability
- SWEBOK www.swebok.org
- SA-based testing

What will you bring home?

- Testing is a complex and expensive task in sw development
- Testing is also an attractive research topic
- Testing is a crucial part of any development methodology (including an SA-based process)
- Although inherently practical, testing can draw great benefits from the usage of formal methods (and vice versa)
- (Formal) SA can usefully drive the testing of large complex systems, by highlighting the architectural relevant “classes” of behavior

Formal Methods in Testing Software Architectures

Overview

Part 1: Testing basics

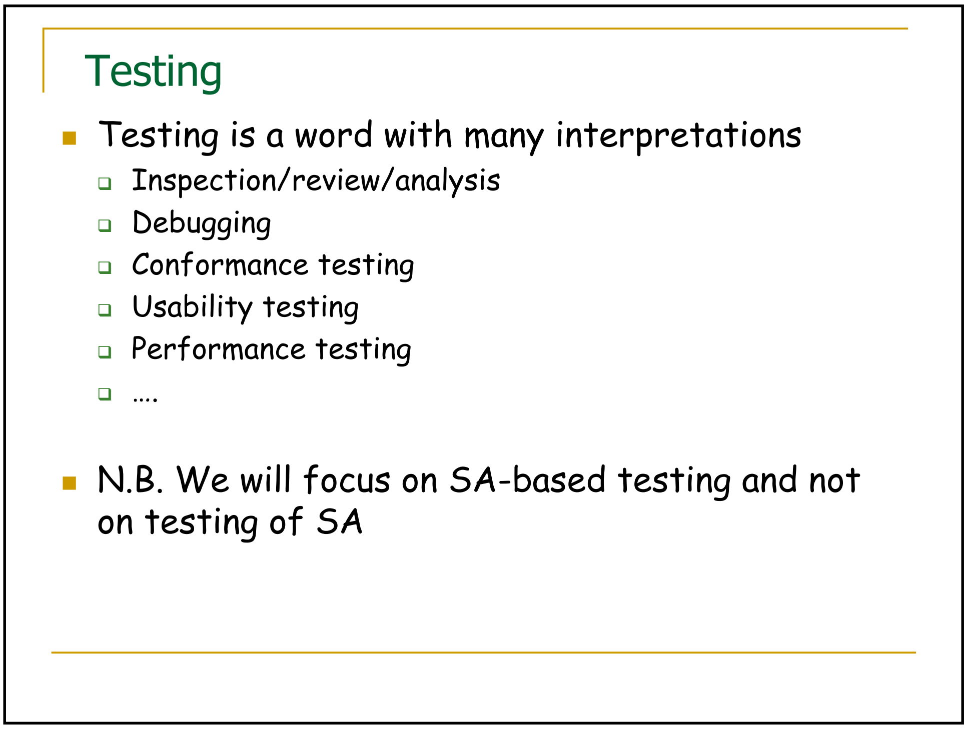
Part 2: (An approach to) SA-based testing

Part 3: Examples and tools

Part 4: A wider perspective

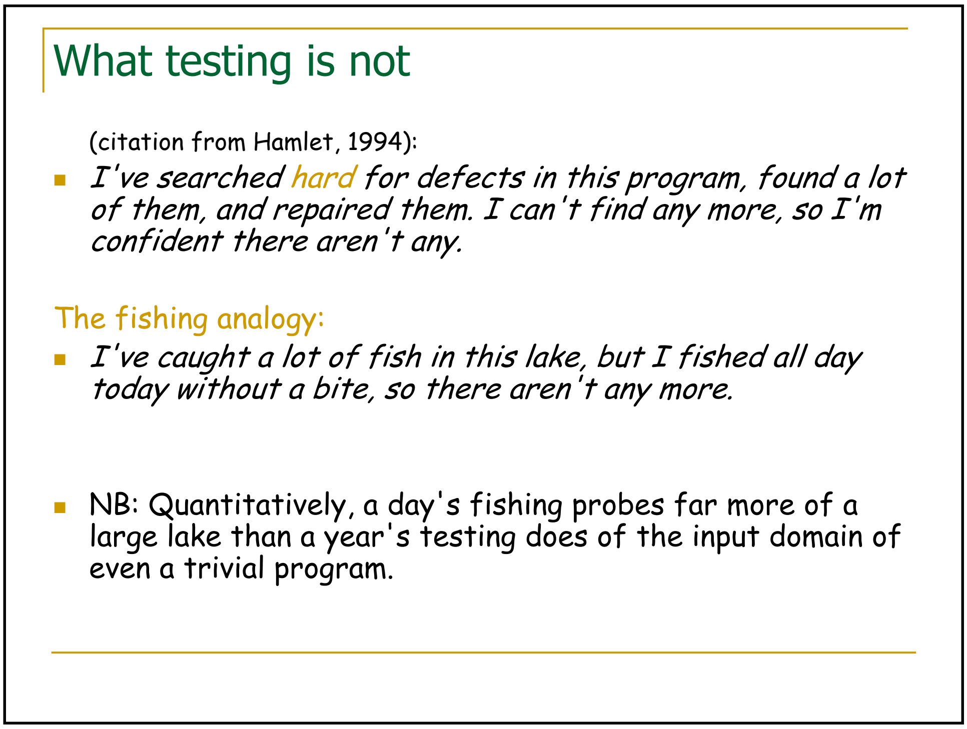
Part 1: Testing basics

- Definition of sw testing
- Fault vs. failure
- Test selection
  - *Coverage testing*
  - *Reliability testing*
  - *Specification-based testing*
- Traceability and test execution
- Testing from LTS



Testing

- Testing is a word with many interpretations
  - Inspection/review/analysis
  - Debugging
  - Conformance testing
  - Usability testing
  - Performance testing
  - ….
- N.B. We will focus on SA-based testing and not on testing of SA



What testing is not

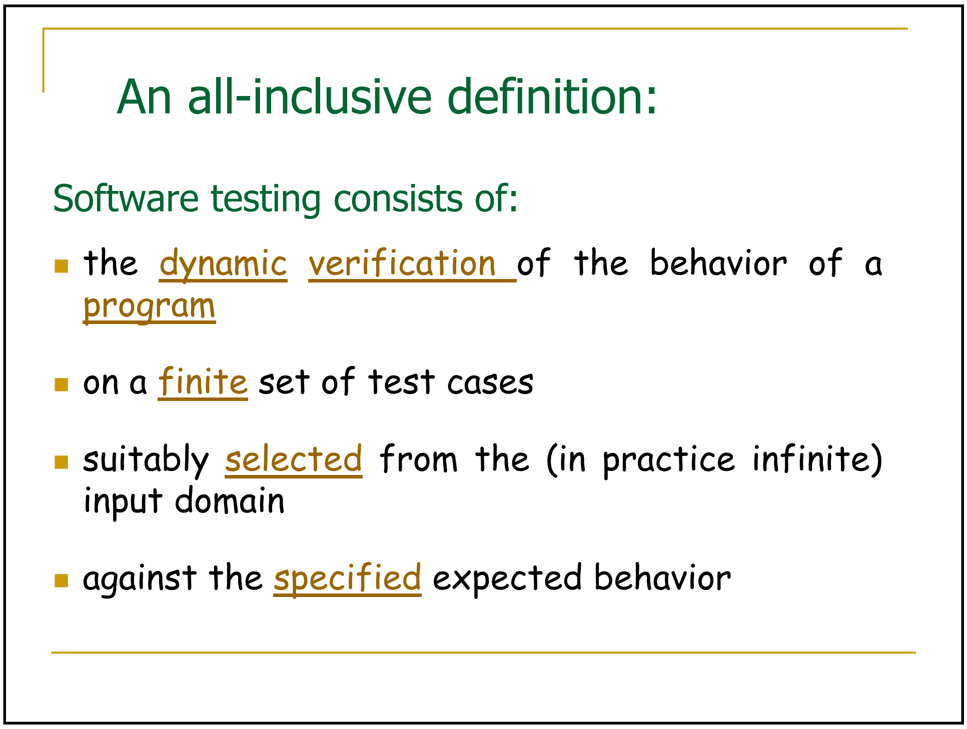
(citation from Hamlet, 1994):

- *I've searched hard for defects in this program, found a lot of them, and repaired them. I can't find any more, so I'm confident there aren't any.*

The fishing analogy:

- *I've caught a lot of fish in this lake, but I fished all day today without a bite, so there aren't any more.*

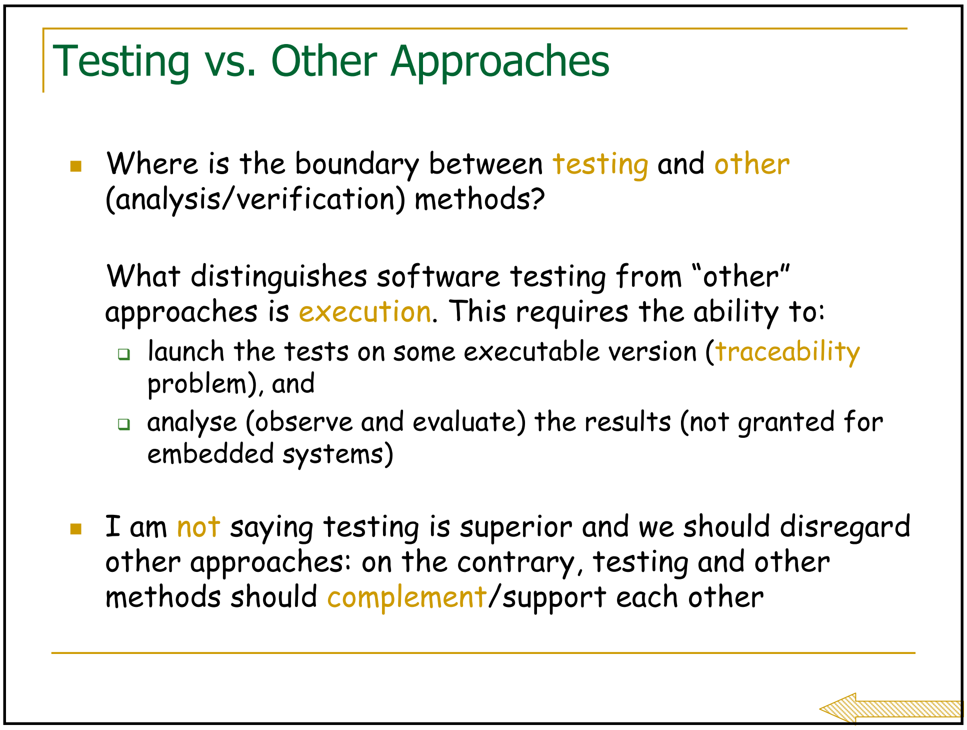
- NB: Quantitatively, a day's fishing probes far more of a large lake than a year's testing does of the input domain of even a trivial program.



An all-inclusive definition:

Software testing consists of:

- the dynamic verification of the behavior of a program
- on a finite set of test cases
- suitably selected from the (in practice infinite) input domain
- against the specified expected behavior



Testing vs. Other Approaches

- Where is the boundary between testing and other (analysis/verification) methods?

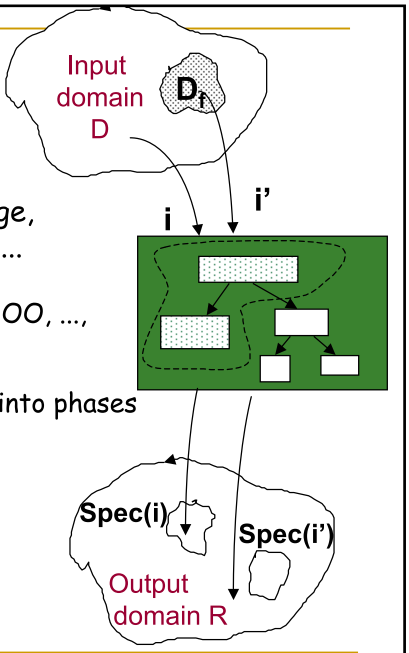
  What distinguishes software testing from "other" approaches is execution. This requires the ability to:
  - launch the tests on some executable version (traceability problem), and
  - analyse (observe and evaluate) the results (not granted for embedded systems)
- I am not saying testing is superior and we should disregard other approaches: on the contrary, testing and other methods should complement/support each other

Test purpose

- The software is tested in order to:
  - Find bugs: “debug testing”, e.g., by:
    - Conformance testing
    - Robustness testing
    - Coverage testing
  - Measure its reliability: by statistical inference
  - Measure its performance
  - Release it to the customer (acceptance test)
  - …

  not mutually exclusive goals, but rather complementary
- Of course different approaches are taken

The System Under Test



- Software systems are increasingly large, complex, distributed, heterogeneous, ….
  - How do we partition the testing of CB, OO, …, systems into manageable pieces?
  - How do we organise the whole process into phases (who, what and how)?
  - Which tests are relevant at a specific phase? (e.g., SA-based tests vs. acceptance tests)
  - Evergreen: how do we regression-test?

The high cost of testing

- Any software test campaign involves a trade-off between
  - Limiting resource utilization (time, effort)
    → as few tests as possible
  - Augmenting our confidence
    → as many tests as possible
- Two research challenges:
  - Determining a feasible and meaningful stopping rule
  - Evaluating test effectiveness (reliability, "coverage notion", ...very tough problem)

(Not-exhaustive) Classification of Test Selection Techniques

"where tests are generated from"

Tester's intuition and expertise	Fault-based
- "Ad hoc testing" (sometime quite effective)	- Error guessing/special cases
- Special cases	- Mutation
Specifications	Usage
- Equivalence partitioning	- SRET
- Boundary-value analysis	- Field testing
- Decision table	Nature of application, e.g.:
- Automated derivation from formal specs (conformance t.)	- Object Oriented
-	- Web
Code	- GUI
- Control-flow based	- Real-time, embedded
- Data-flow based	- Scientific
	- .....

No one is the best technique, but a combination of different criteria has empirically shown to be the most effective approach

The saturation effect

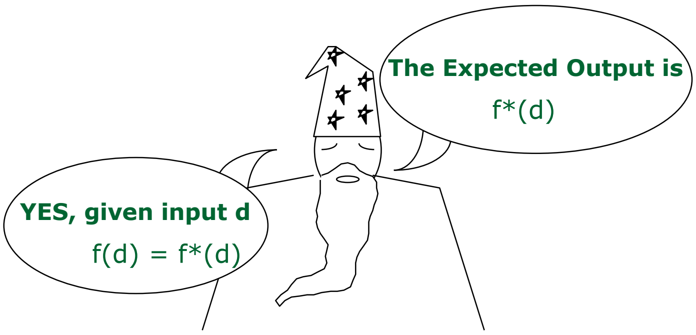
- Every test criterion has a limit on the number of failures it can detect.
- The saturation effect leads to overestimation of reliability: because of the saturation effect, the estimated reliability increases while the true reliability remains constant.
- Different criteria find different kind of faults

===> Combining different test criteria is more efficient than any test criterion used in isolation

Coverage analysis on two programs thoroughly tested for several years based on functionality showed:

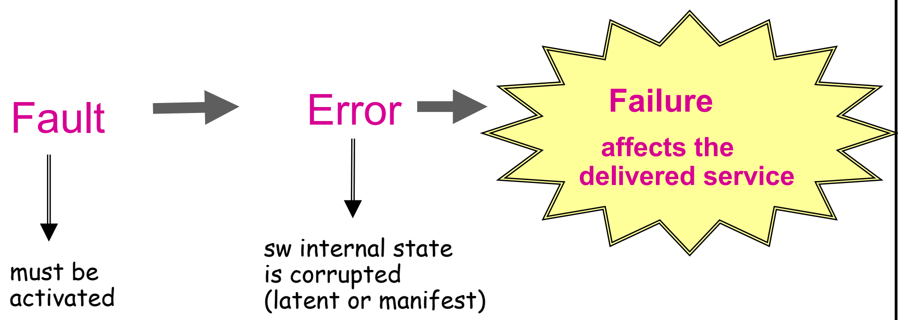
(70 / 85) branch, (48 / 53) p-use, (55 / 48) c-use

The oracle problem



- In some cases easier (e.g., an existing version, existing formal specification), but generally very difficult (e.g., operational testing)
- Not enough emphasized research problem?

THE FAULT-FAILURE MODEL



Fault

must be activated

Error

sw internal state is corrupted (latent or manifest)

Failure

affects the delivered service

A fault will manifest itself as a failure if all of the 3 following conditions hold:

1) EXECUTION: the faulty piece of code is executed

2) INFECTION: the internal state of the program is corrupted: *error*

3) PROPAGATION: the error propagates to program output

PIE MODEL (Voas, IEEE TSE Aug. 1992)

The ambiguous notion of a "fault"

- What a tester or a user observes is a failure. What is then a fault?
    - It is a useful and intuitive notion, but very difficult to formalize.
- In practice we often characterise a fault with the modifications made to fix it: e.g. "Wrong operator". In such a way,
    - the concept of a fault becomes meaningful only after it has been fixed.
    - Moreover, different people may react to a same failure with different fixes (and clever programmer find minimum fixes)
- A solution is to avoid the ambiguous notion of a fault and reason instead in terms of failure regions, i.e. collections of input points that give rise to failures.
    - A failure region is characterised by a size and a pattern

A hot debate

Partition(i.e., systematic) vs. (Uniform) Random testing:

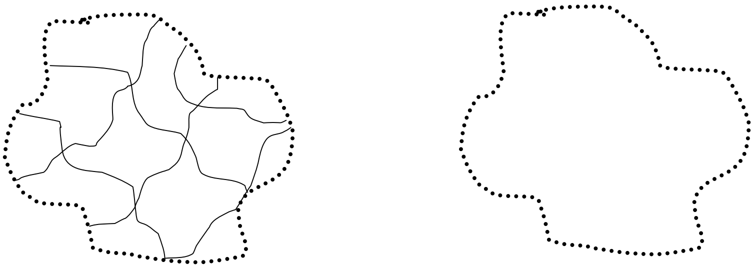
- Ideally a testing criterion:

  C(Program, Reference Model, Test Set)

  induces a subdivision of the input domain into disjoint classes of inputs exposing an equivalent behavior:
  - in this way a test within each partition is sufficient to check the behavior for the whole (test hypothesis)
- In practice:
  - Overlapping Sub-domains
  - Not homogeneous behaviour

Partition vs. Random testing

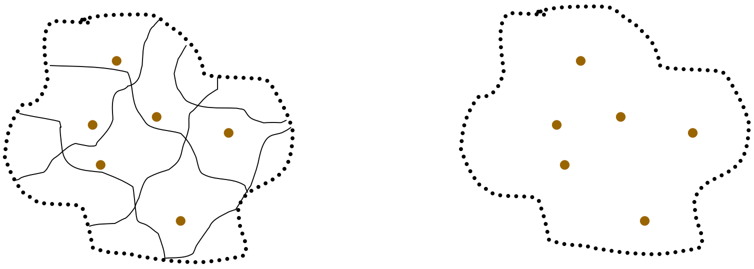
Partition vs. Random testing



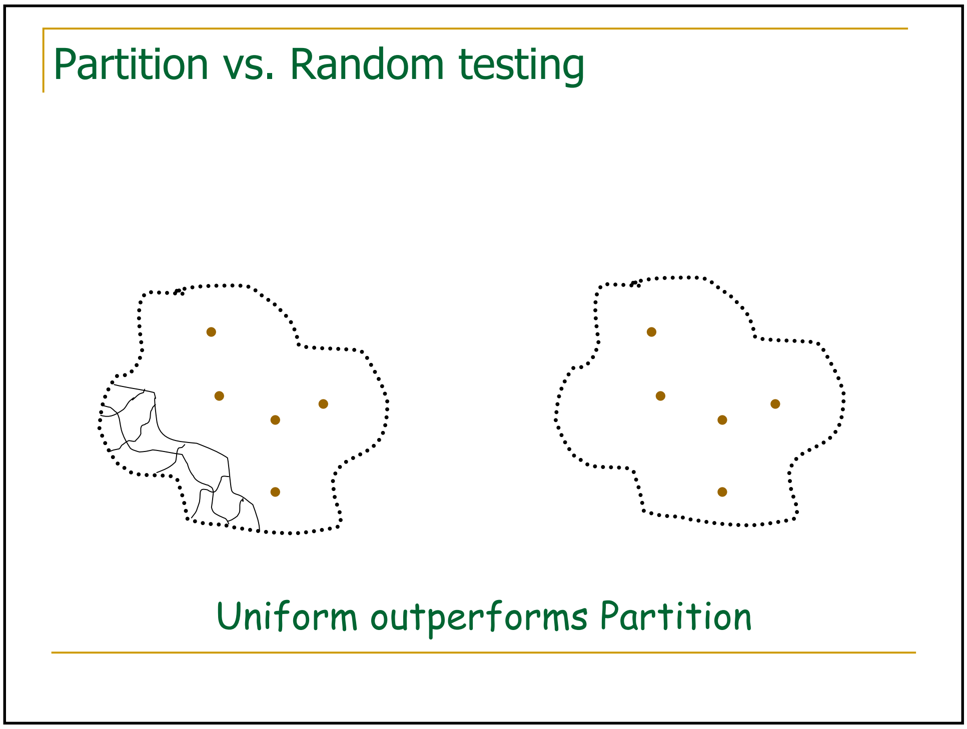
$P_p = 1 - \Pi\,(1-\vartheta_j\,)$ $P_r = 1 - \,(1-\vartheta)^n$

where, for a fair comparison, an equal number of test cases is assumed

Partition vs. Random testing

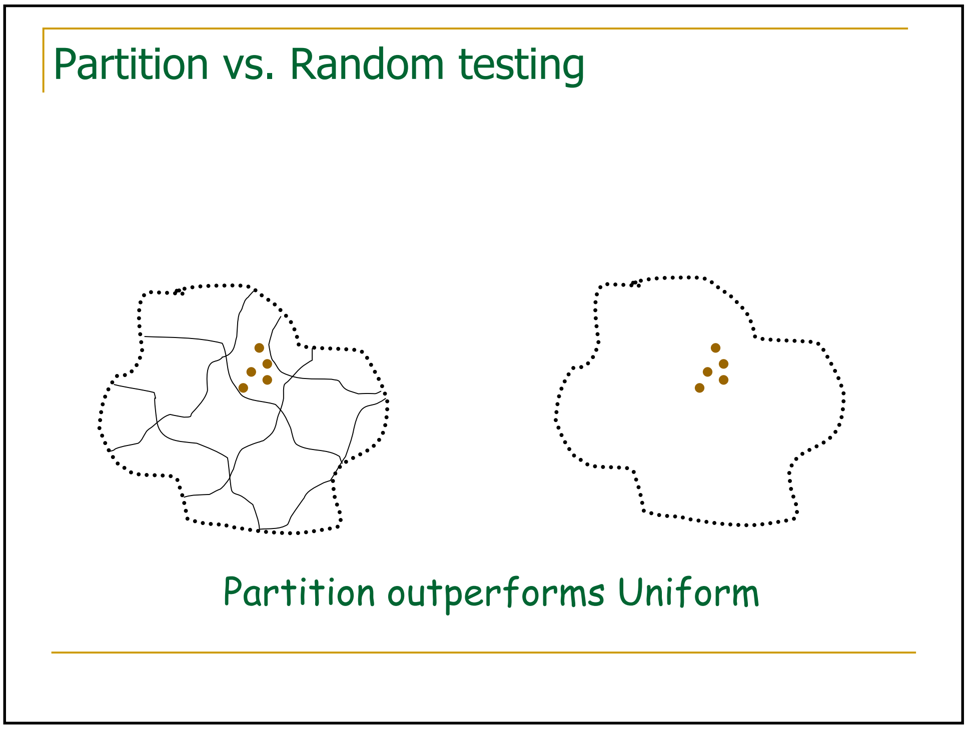


~ Equivalent case



Partition vs. Random testing

Uniform outperforms Partition



Partition vs. Random testing

Partition outperforms Uniform

Use of a Test Criterion

C(Program, Reference Model, Test Set)

For deriving the Test Set:

- (automated) test selection/generation

For deciding whether a Test Set is sufficient:

- stopping rule/adequacy criterion

Elements of a Test Criterion

C(Program, Reference Model, Test Set)

Program

- A Procedure/function
- a Class
- a Package
- a Component

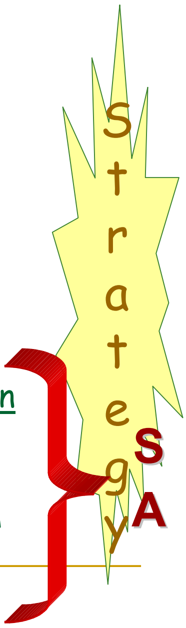
} Unit testing

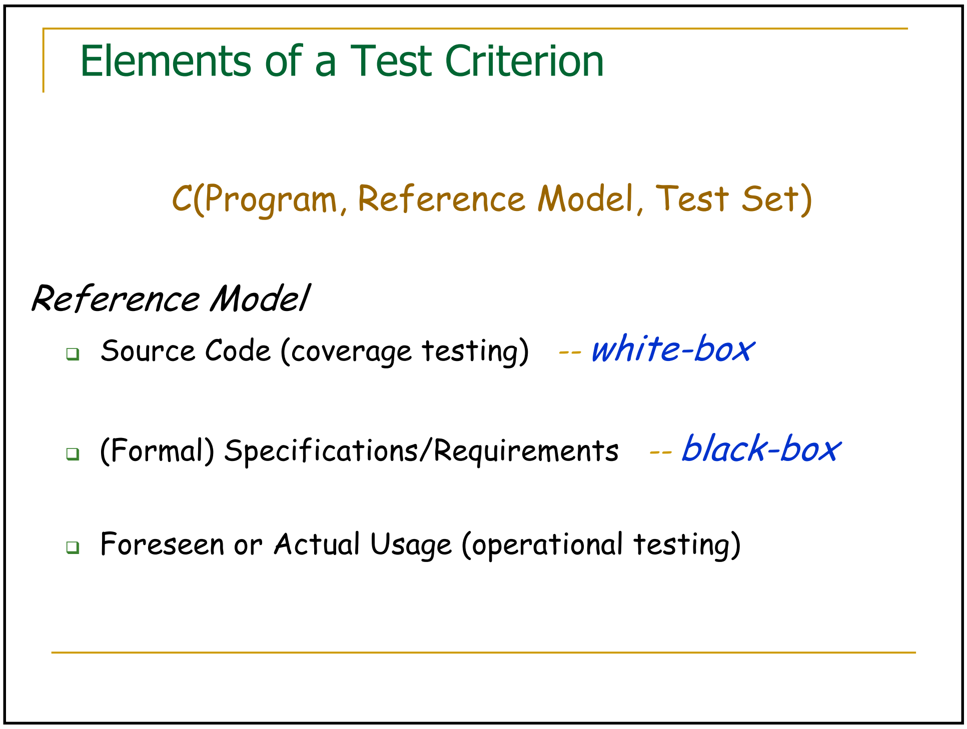
- Putting some pieces together (classes, packages, components)

} Integration testing

- the whole system on a host machine
- the whole system on the target

} System testing



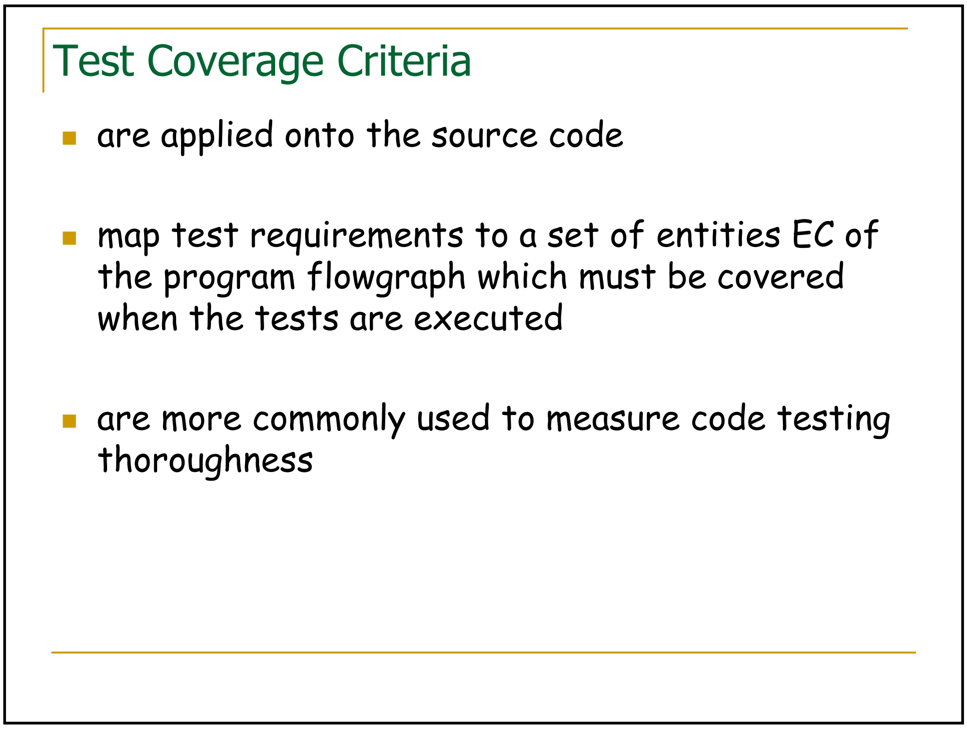


Elements of a Test Criterion

C(Program, Reference Model, Test Set)

Reference Model

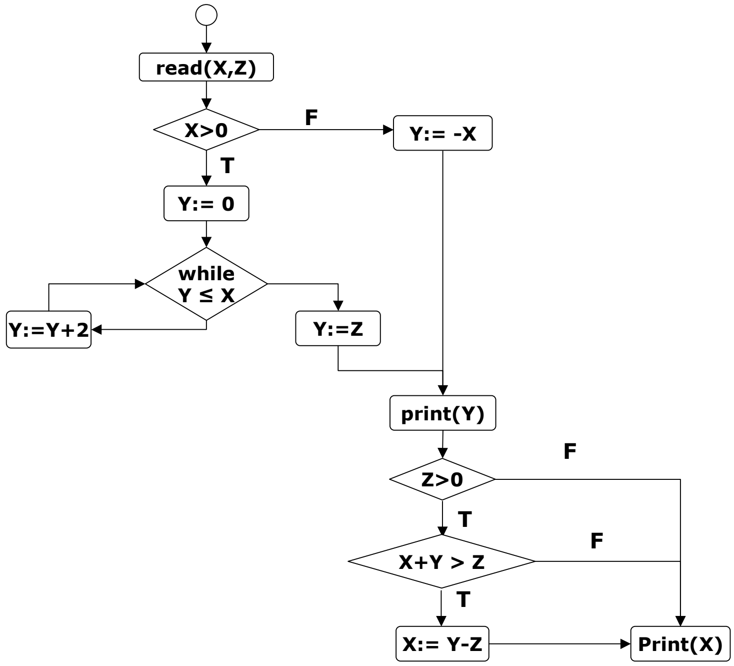
- Source Code (coverage testing) -- *white-box*
- (Formal) Specifications/Requirements -- *black-box*
- Foreseen or Actual Usage (operational testing)



Test Coverage Criteria

- are applied onto the source code
- map test requirements to a set of entities EC of the program flowgraph which must be covered when the tests are executed
- are more commonly used to measure code testing thoroughness

The “Traditional” Flowgraph



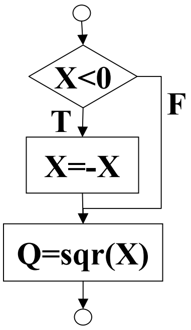
e.g., Branch Adequacy Criterion

- requires each program branch to be executed by at least one test case
- A test suite T satisfies the Branch adequacy criterion for a program P iff, for each branch B of P, there is one test case in T that causes the execution of B
- NB: Branch coverage is not guaranteed by Statement coverage

It should be the minimum mandatory requirement at Unit Test level

(Counter-)Example: Branch coverage vs. Any Nonempty Input Subset (ANIS)

```
Function Q (x:real): real;
 {should compute sqrt |x|}
  begin
    if x<0 then x:= -x;
    Q:= sqr(x)
 end;
```



Branch testing subsumes ANIS, yet
[1, -1] achieves branch coverage and uncovers no failures

[2] (does not even cover all branches and) reveals the failure

Coverage basic concepts

- Coverage is a measure of completeness
- 100% coverage never means “complete testing”, but only completeness wrt the selected strategy
- Several strategies and coverage metrics
- No best one, but some (on average) better than others

Operational Testing

A completely different approach:

We cannot realistically presume to find and remove the last failure.

Then, we should invest test resources to find out the "biggest" ones.

Testing under the operational profile

- "Goodness" here means to minimise the user perceived faults, i.e.
  - Try to find earlier those bugs that are more frequently encountered (neglect "small" bugs)
  - Not suitable for safety critical applications

Operational Testing

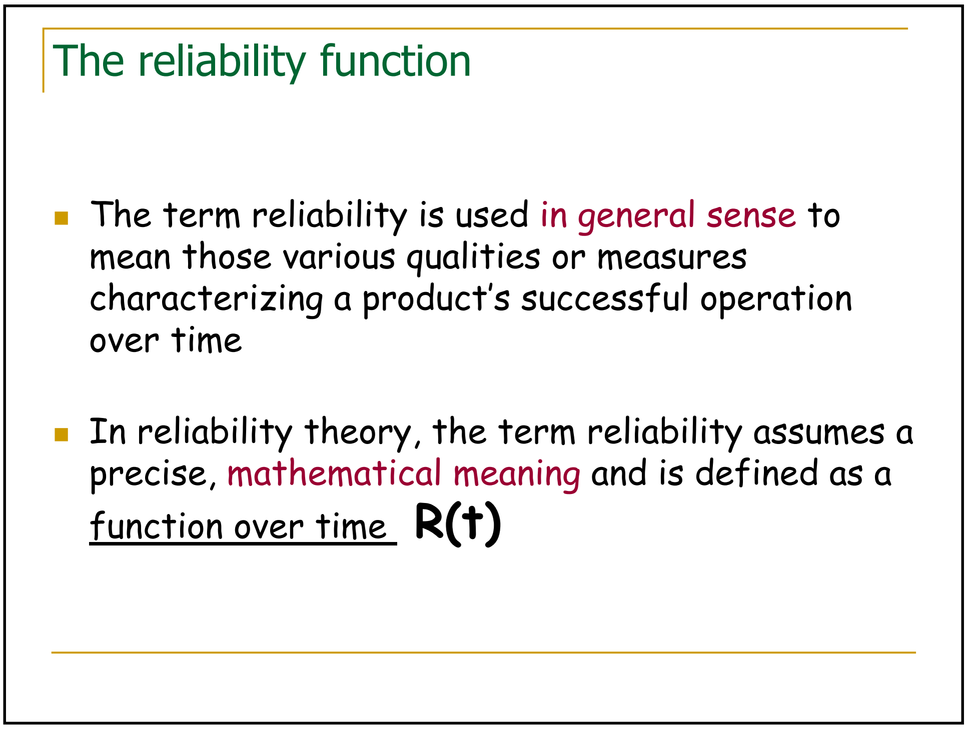
A completely different approach:

We cannot realistically presume to find and remove the last failure.

Then, we should invest test resources to find out the "biggest" ones.

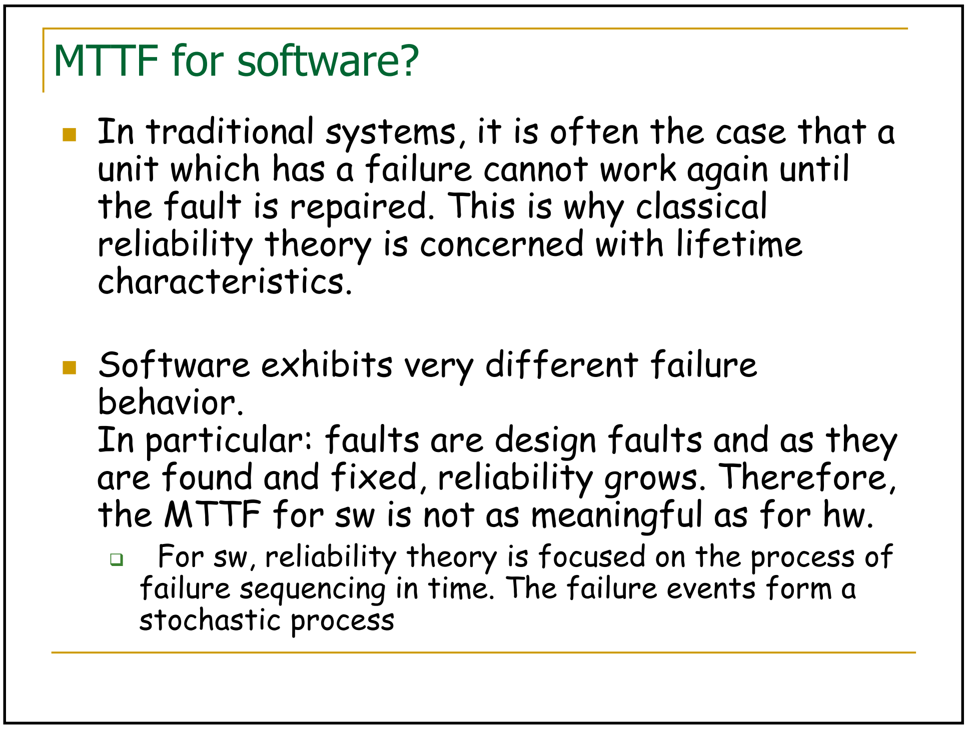
The importance of reliability

- In today's industrial production, **reliability** is a strategic product variable, as important as cost and performance
- For example, when we select a car, not only we compare price, comfort, speed, but also repair records.
  - i.e., wrt reliability we require products to prove that they can work without failing for "enough" long time.



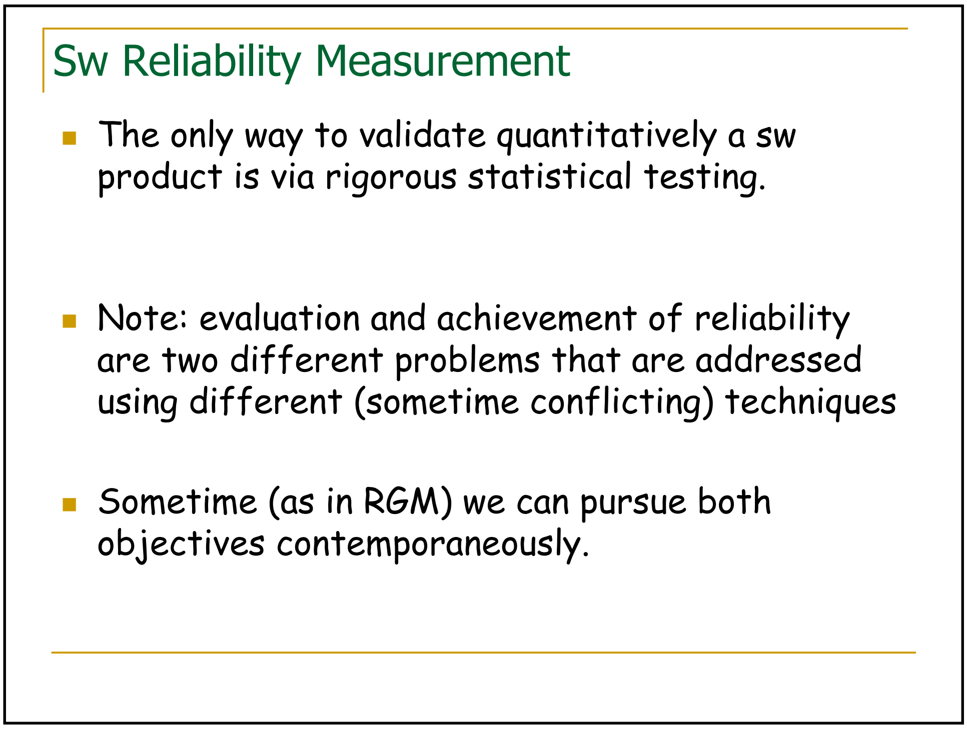
The reliability function

- The term reliability is used in general sense to mean those various qualities or measures characterizing a product’s successful operation over time
- In reliability theory, the term reliability assumes a precise, mathematical meaning and is defined as a function over time **R(t)**



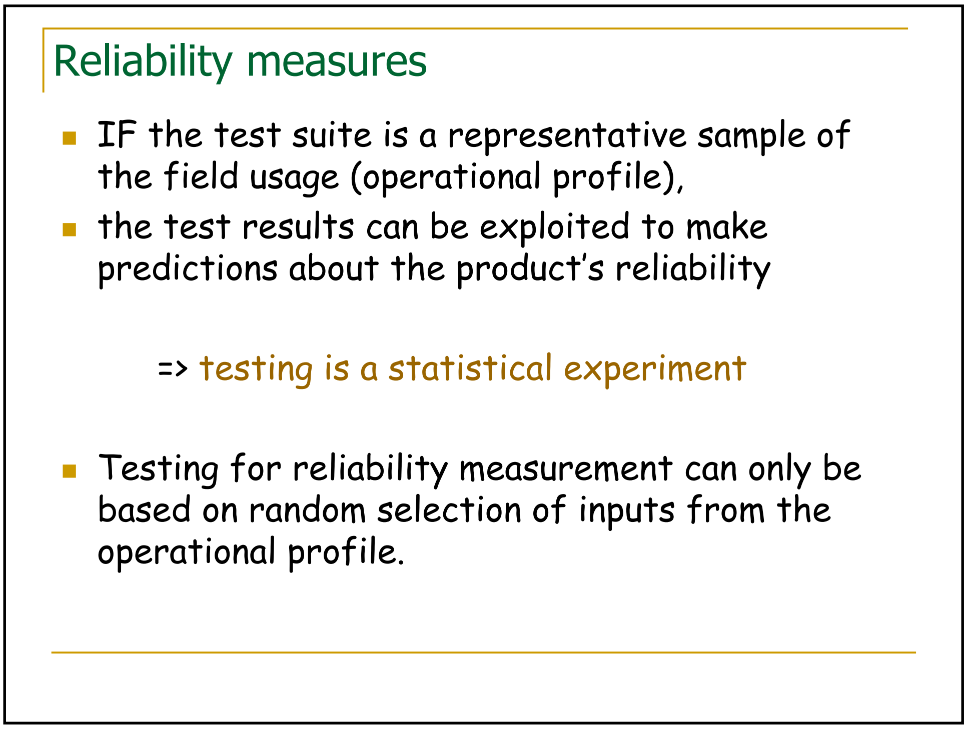
MTTF for software?

- In traditional systems, it is often the case that a unit which has a failure cannot work again until the fault is repaired. This is why classical reliability theory is concerned with lifetime characteristics.
- Software exhibits very different failure behavior.
  In particular: faults are design faults and as they are found and fixed, reliability grows. Therefore, the MTTF for sw is not as meaningful as for hw.
  - For sw, reliability theory is focused on the process of failure sequencing in time. The failure events form a stochastic process



Sw Reliability Measurement

- The only way to validate quantitatively a sw product is via rigorous statistical testing.
- Note: evaluation and achievement of reliability are two different problems that are addressed using different (sometime conflicting) techniques
- Sometime (as in RGM) we can pursue both objectives contemporaneously.



Reliability measures

- IF the test suite is a representative sample of the field usage (operational profile),
- the test results can be exploited to make predictions about the product's reliability

=> testing is a statistical experiment

- Testing for reliability measurement can only be based on random selection of inputs from the operational profile.

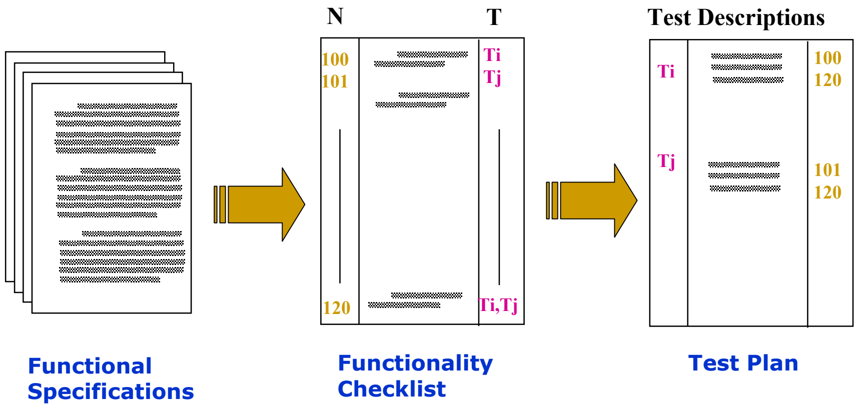
Components of sw reliability

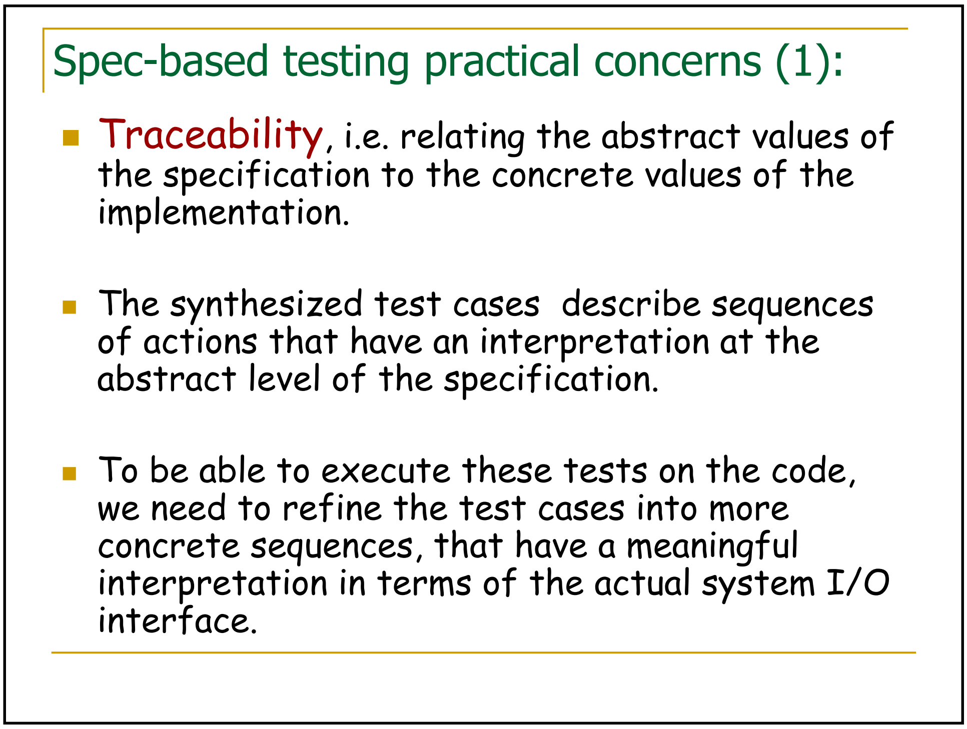
Sw reliability measurement is a new technology (pioneering work by Musa).

The essential components are:

1. Establish reliability goals
2. Develop operational profile
3. Execute tests
4. Interpret test results

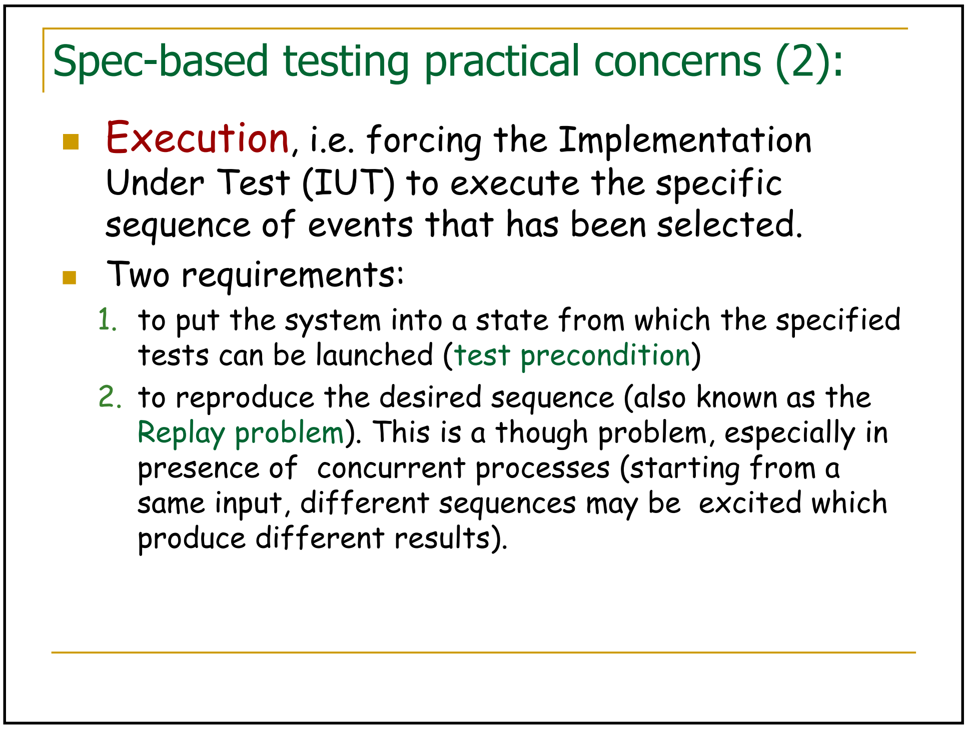
Specification-based Testing





Spec-based testing practical concerns (1):

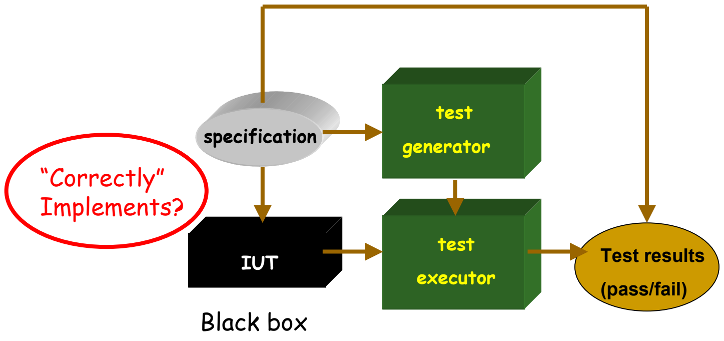
- Traceability, i.e. relating the abstract values of the specification to the concrete values of the implementation.
- The synthesized test cases describe sequences of actions that have an interpretation at the abstract level of the specification.
- To be able to execute these tests on the code, we need to refine the test cases into more concrete sequences, that have a meaningful interpretation in terms of the actual system I/O interface.



Spec-based testing practical concerns (2):

- Execution, i.e. forcing the Implementation Under Test (IUT) to execute the specific sequence of events that has been selected.
- Two requirements:
  1. to put the system into a state from which the specified tests can be launched (test precondition)
  2. to reproduce the desired sequence (also known as the Replay problem). This is a though problem, especially in presence of concurrent processes (starting from a same input, different sequences may be excited which produce different results).

Conformance Testing



With formal specifications:

- algorithmic generation of tests
- formal validation of test results

Formal Testing (not-exhaustive references!!!)

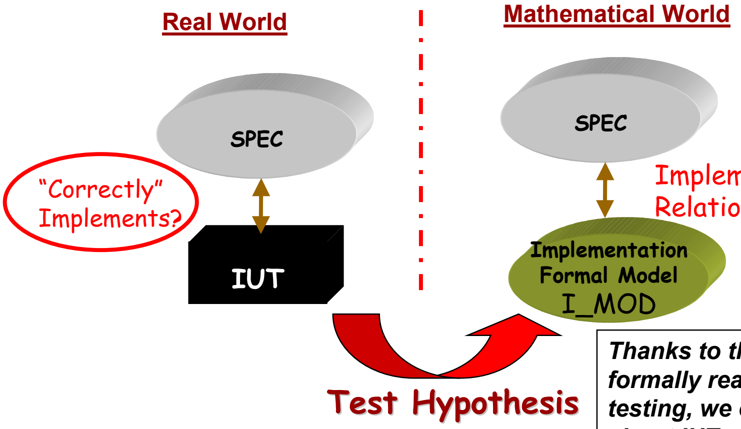
Based on dynamic behavior:

- FSM-based (Bochmann&Petrenko, Lee&Yannakis, …)
- LTS-based (Brinksma&Tretmans, Jard&Jeron …)

Focussing on static aspects,

- ADT theory (Bernot&Gaudel…)
- Z-based (Hierons…)

Conformance relation



Thanks to the Test Hyp. to formally reason about testing, we can forget about IUT and consider Mod

IUT is "correct" (conforms_to) SPEC

iff MOD is *Imp*-related to SPEC

The *Imp*-relation

- Intuitively, this relation links any observation we can make of I_MOD within an environment to an observation of SPEC in the same environment

- For instance:
  - LTSs as observers
  - Set inclusion as link
  - Traces as observations

- Several *Imp*-relations, originated to establish equivalence, or preorder, relations between transition systems.

The ioco relation (Tretmans)

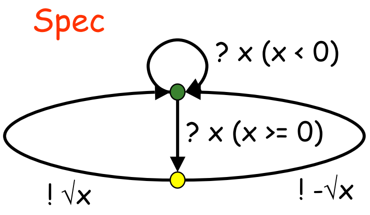
- Distinguishes Input and Output communications exchanged between the implementation and the observer (test), i.e.:
  - Controllable events
  - Observable events

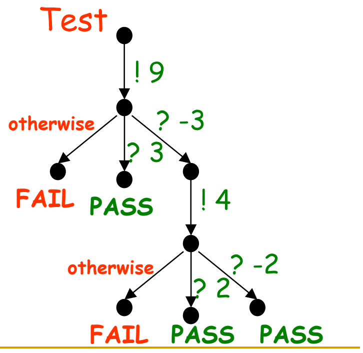
I_Mod is **ioco**-correct wrt Spec iff

1. if I_Mod produces output *out* after a trace σ, then Spec can produce *out* after σ
2. if I_Mod cannot produce any output after trace σ, then Spec cannot produce any output after σ (absence of output or "quiescence")

Formal test case

- specifies the behaviour of a tester (the observer of the implementation)
- Again, it is formally defined as an IOLTS (must be deterministic and have finite behavior), and contains terminal states: Pass and Fail.





The ideal case

An ideal test suite $\mathcal{T}$ should guarantee that an implementation passes $\mathcal{T}$ if and only if it conforms.

An ideal test suite holds the two properties of:

- Soundness: test will never fail with **ioco**-correct implementation
  I_Mod **ioco** Spec implies I_Mod passes T
- Exhaustiveness: for each **ioco**-incorrect implementation a test can be generated that detects it
  I_Mod **ioco** Spec implies ∃T such that I fails T

Test generator validity

Every set of tests T generated with the algorithm should yield soundness and exhaustiveness, but the latter requires infinite tests, so in practice only the former is guaranteed.

Sound test suite can only detect non-conformance, but cannot assure conformance.

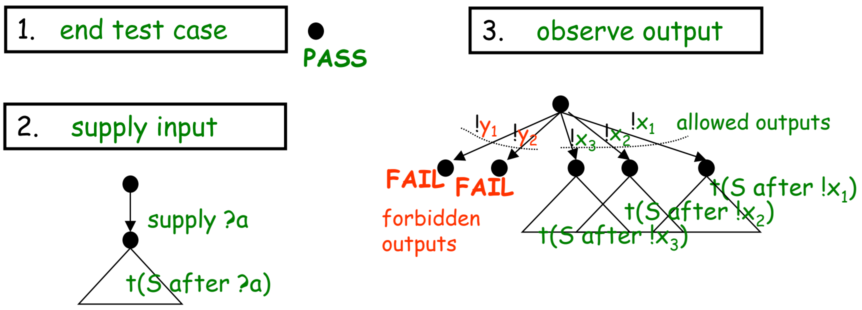
Exhaustiveness is proved for the set of all test cases that could be generated.

Test Generation in TorX

Nondeterministic algorithm

Generate a test case t(S) from a transition system model with S a set of states (initially S = {s_0})

Apply the following steps recursively, nondeterministic choice :



1. end test case

PASS

2. supply input

supply ?a

t(S after ?a)

3. observe output

$!y_1$ $!y_2$ $!x_3$ $!x_2$ $!x_1$ allowed outputs

FAIL FAIL

forbidden outputs

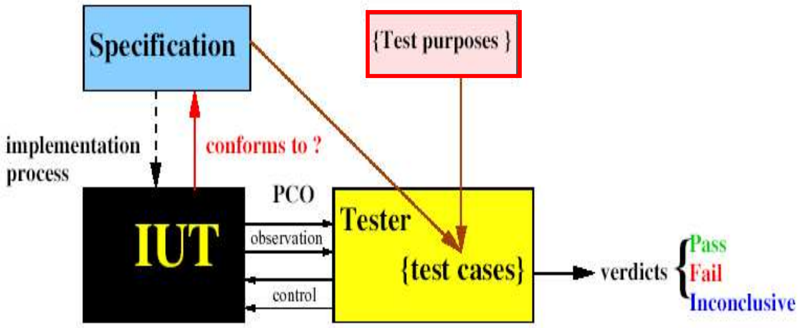
t(S after $!x_1$)

t(S after $!x_2$)

t(S after $!x_3$)

TGV-based formal testing

[T.Jeron, Irisa]



Specification

{Test purposes }

implementation process

conforms to ?

PCO

IUT

observation

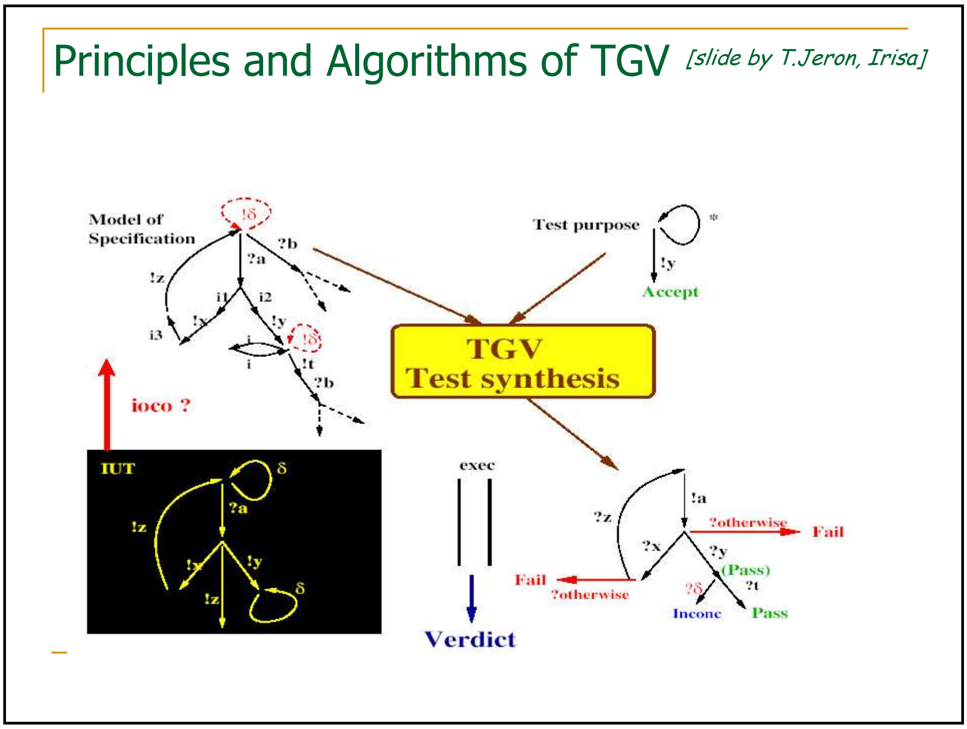
control

Tester

{test cases}

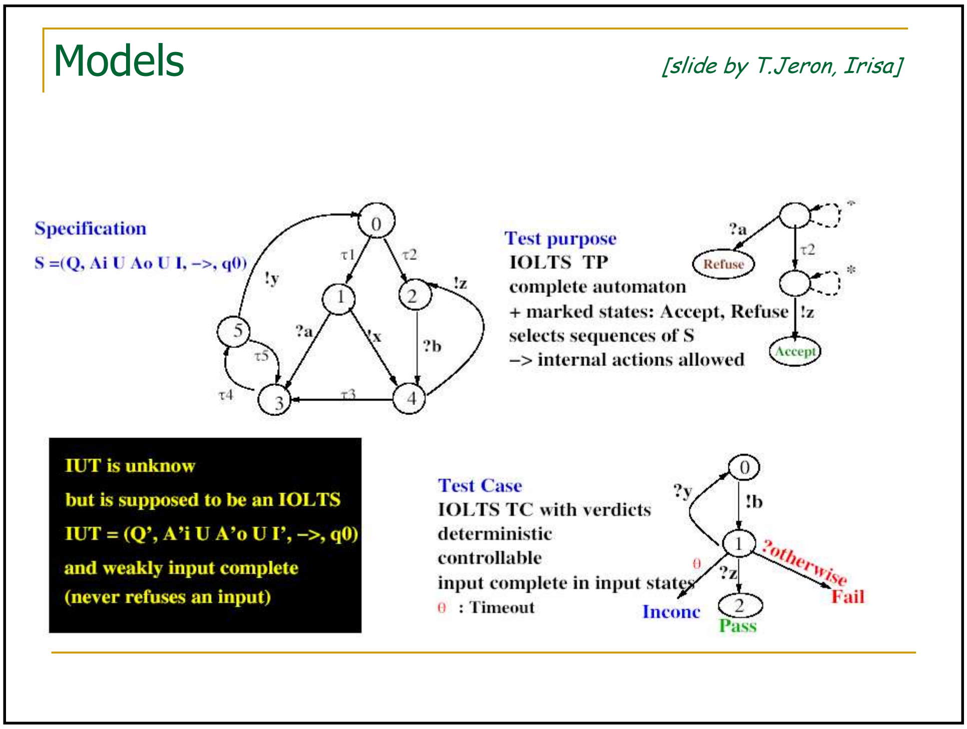
verdicts {Pass Fail Inconclusive

⇒ automatic test synthesis from formal specifications and test purposes



Principles and Algorithms of TGV

[slide by T.Jeron, Irisa]



Models

[slide by T.Jeron, Irisa]

Specification

S =(Q, Ai U Ao U I, –>, q0)

Test purpose

IOLTS TP

complete automaton

+ marked states: Accept, Refuse

selects sequences of S

–> internal actions allowed

IUT is unknow

but is supposed to be an IOLTS

IUT = (Q', A'i U A'o U I', –>, q0)

and weakly input complete

(never refuses an input)

Test Case

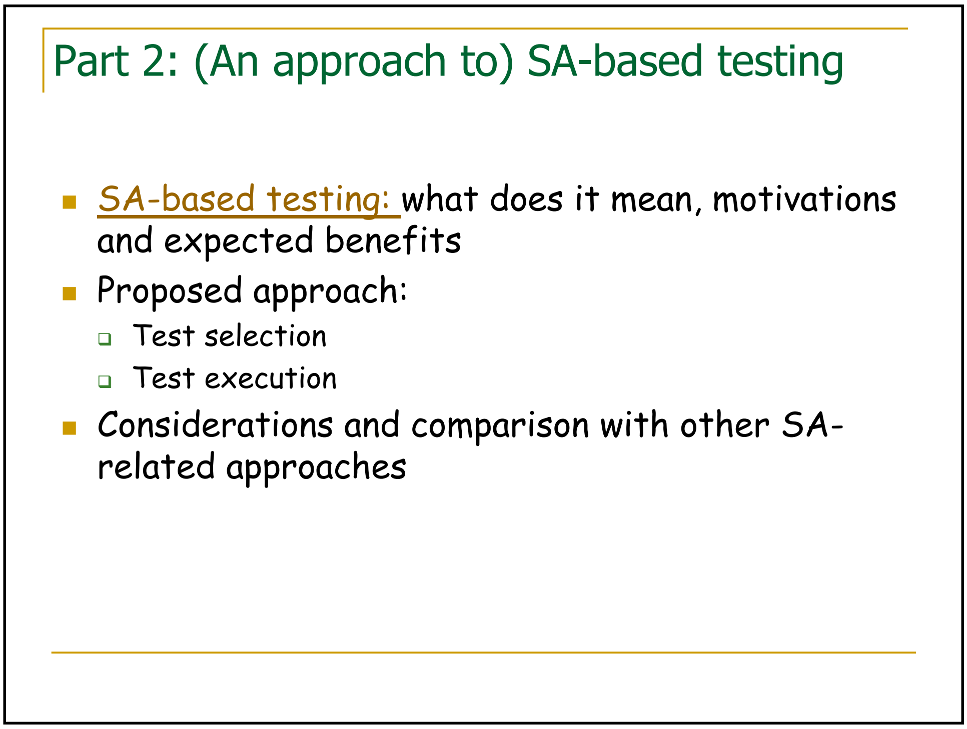
IOLTS TC with verdicts

deterministic

controllable

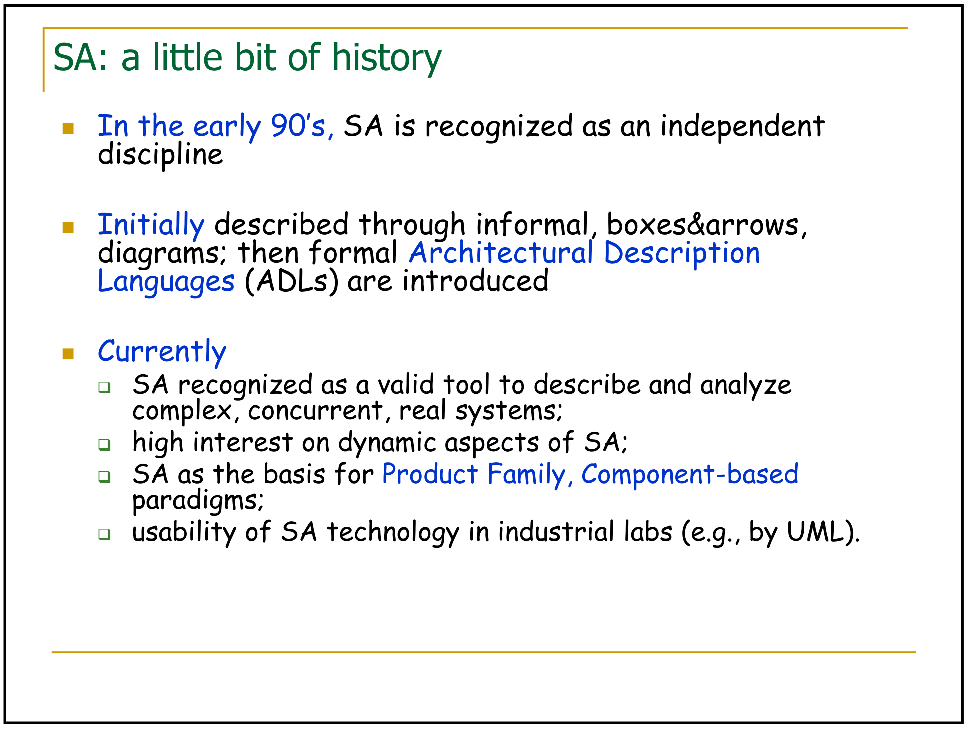
input complete in input states

θ : Timeout



Part 2: (An approach to) SA-based testing

- SA-based testing: what does it mean, motivations and expected benefits
- Proposed approach:
  - Test selection
  - Test execution
- Considerations and comparison with other SA-related approaches



SA: a little bit of history

- In the early 90's, SA is recognized as an independent discipline
- Initially described through informal, boxes&arrows, diagrams; then formal Architectural Description Languages (ADLs) are introduced
- Currently
  - SA recognized as a valid tool to describe and analyze complex, concurrent, real systems;
  - high interest on dynamic aspects of SA;
  - SA as the basis for Product Family, Component-based paradigms;
  - usability of SA technology in industrial labs (e.g., by UML).

In general terms…

- SA (formally) describes how a system is organized into components and connectors…

⇓

- Components
  - computation
- Connectors
  - interaction
- Ports, Interfaces, …

SA Structure (topology)

…and how these components interact

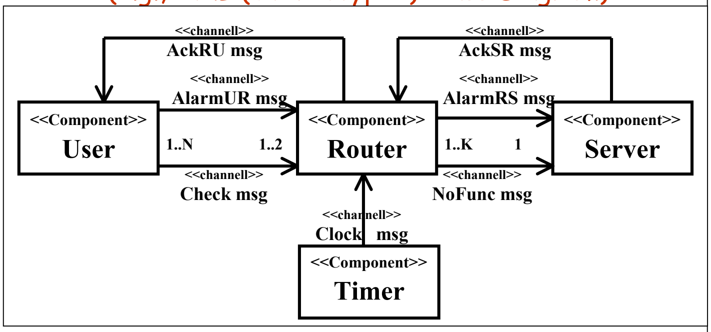
SA Dynamics (behavior)

Software Architecture: Static Description

Static View

Diagrams (UML)

(e.g., UML (Stereotyped) Class Diagram)

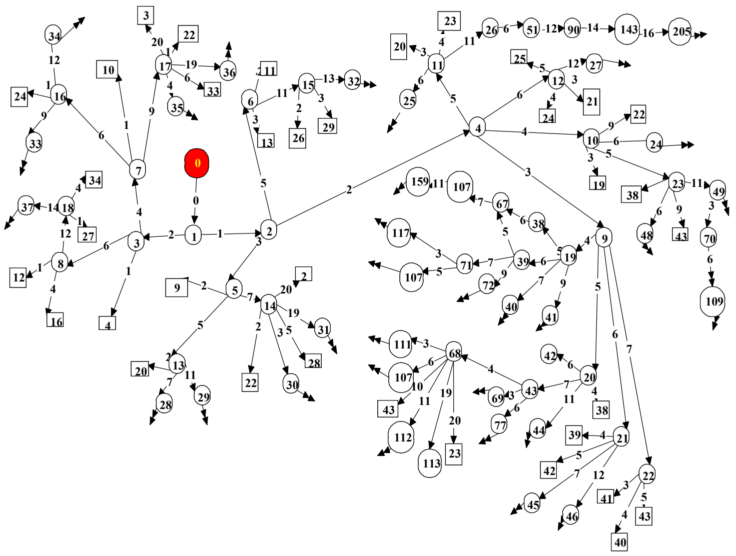


SA dynamics

- A model of Software Architecture behavior:
  - Labeled Transition System (LTS)
  - Finite State Machine
  - Petri Nets
  - …

- Given the SA formal description, the model can be automatically generated; the dynamics is expressed in terms of *components interactions via connectors*

SA: Dynamic Description



Several ADLs

- Chemical Abstract Machine (Cham) ☛ Reactions
- Darwin & FSP ☛ Distributed systems
- Rapide ☛ Events
- Wright ☛ Behavioral Properties
- C2SADL ☛ Dynamic Systems
- Koala ☛ Product Families
- xADL1.0 ☛ Implementation mappings
- ...

No pre-selected choice…

Why SA

- To cope with real world systems (concurrency, complexity, …) using a *high level of Abstraction*
- Compositionality
- *Static* and *dynamic* views
- To discover errors *as early as possible* (e.g., deadlocks)
- To *drive* software system management and evolution

Some early citations

- Perry and Wolf, '92: [PerryWolf92]
  - "*Architecture* is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design."
  - elements are divided into processing elements, data elements and connection elements
- Garlan and Shaw, '93: [GarlanShaw93]
  - SA for a specific system can be captured as ``a collection of computational components - or simply components - together with a description of the interactions between these components - the connectors -"

Why SA (2)

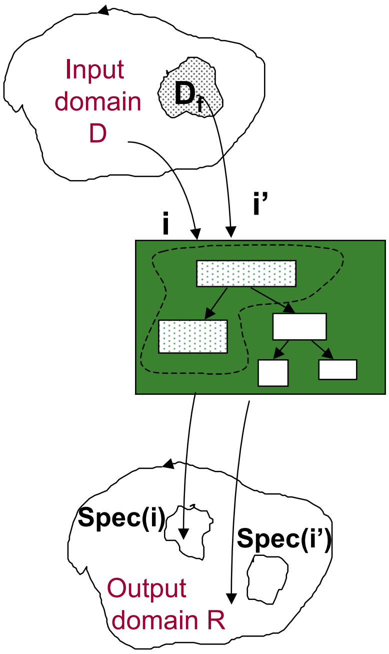
For Analysis throughout an SA-based development process:

SA management is expensive and time consuming

- ⇒ **maximize the benefits**
- ⇒ **analyze SA as much as possible**

- **ADL- based analysis of SA**
- **Model checking SA**
- **SA and Coordination models**
- **Deadlock detection on SA-based specification**
- ***SA and Testing***
- **SA and Performance Analysis**
- **SA and Security**

The System Under Test



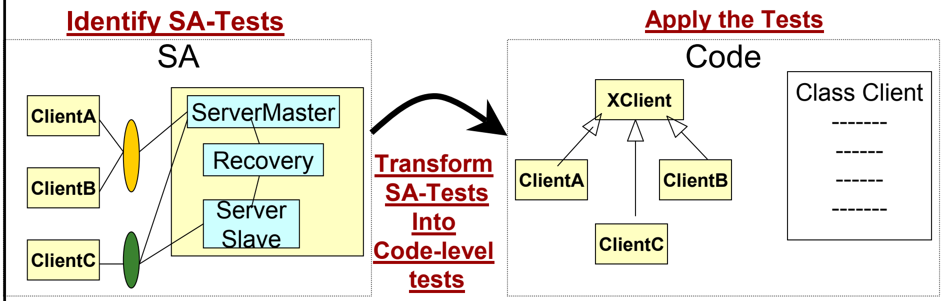
Architecture-based Integration Testing

- **Integration Testing:** unit-tested components are integrated to build complex systems
  - Integration Testing is aimed at exposing problems in the interfaces and interactions between the system components
- **Functional**: focus on the functional requirements
- **Architectural**: information for testing is extracted from the Architectural specification…
- **Down to the Code**: … and propagated down to the Code

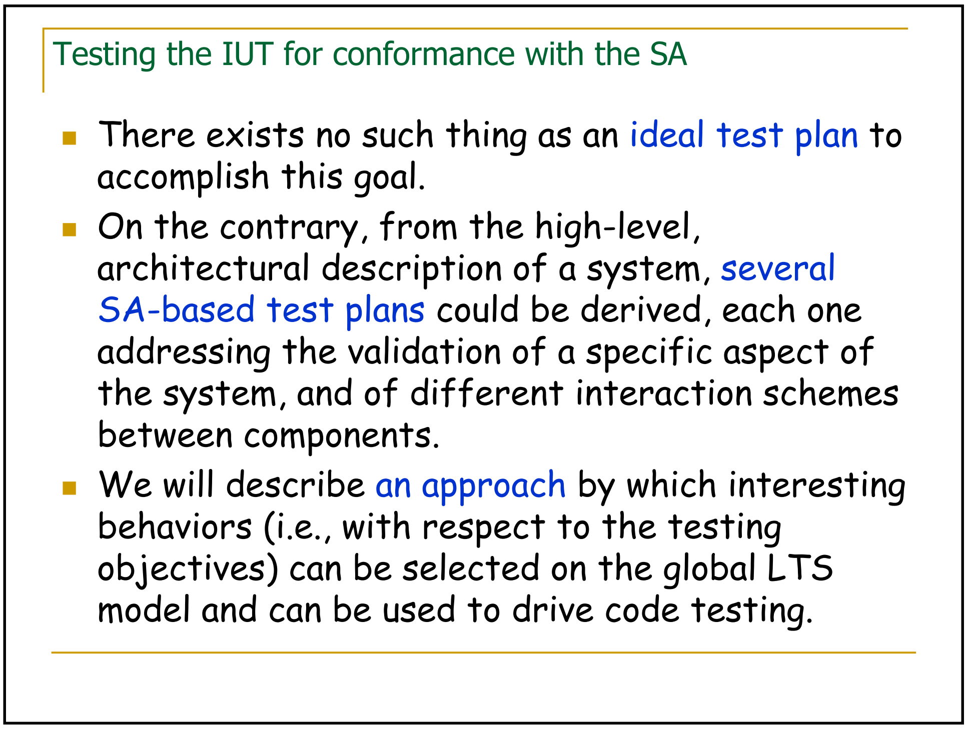
Architecture-based Integration Testing: Expected Benefits

- Test planning interwoven with development and evolution (from Requirements to Code)
- To detect and remove problems as early as possible
- High reusability
- To handle complexity in the testing of large systems
- To combine Structural and Specification-based Testing
- SA-based Integration Testing:
  - Only components interactions…
  - … at an appropriate level of abstraction
  - CBSE and COTS

The main idea…

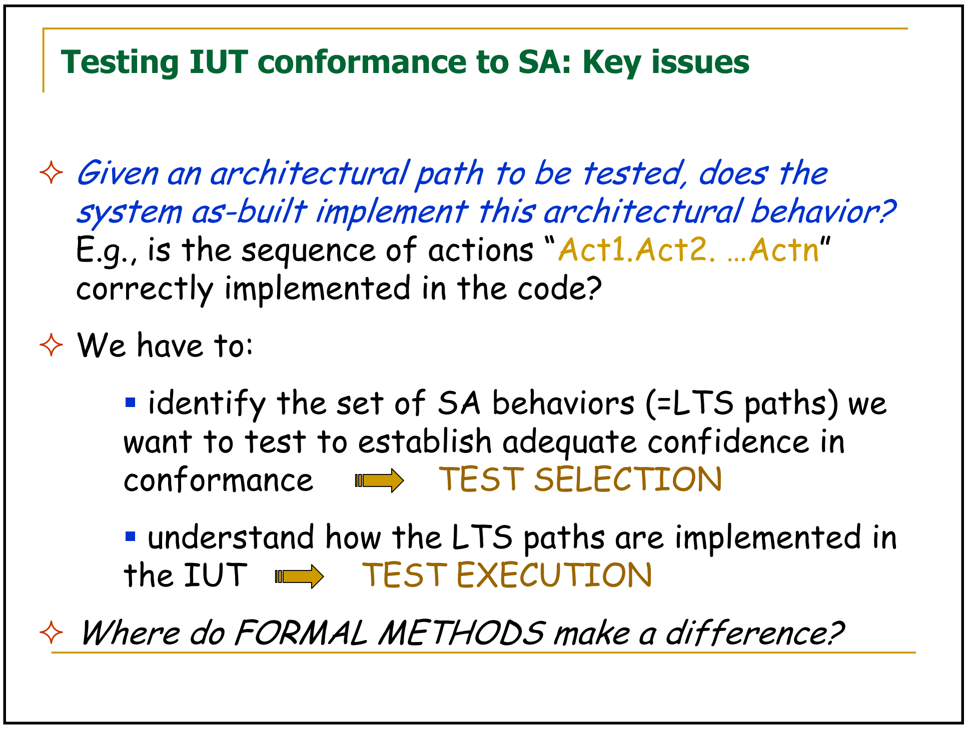


- Goal:
  - Derive test cases
  - using software architectural description
  - run test cases on the Code



Testing the IUT for conformance with the SA

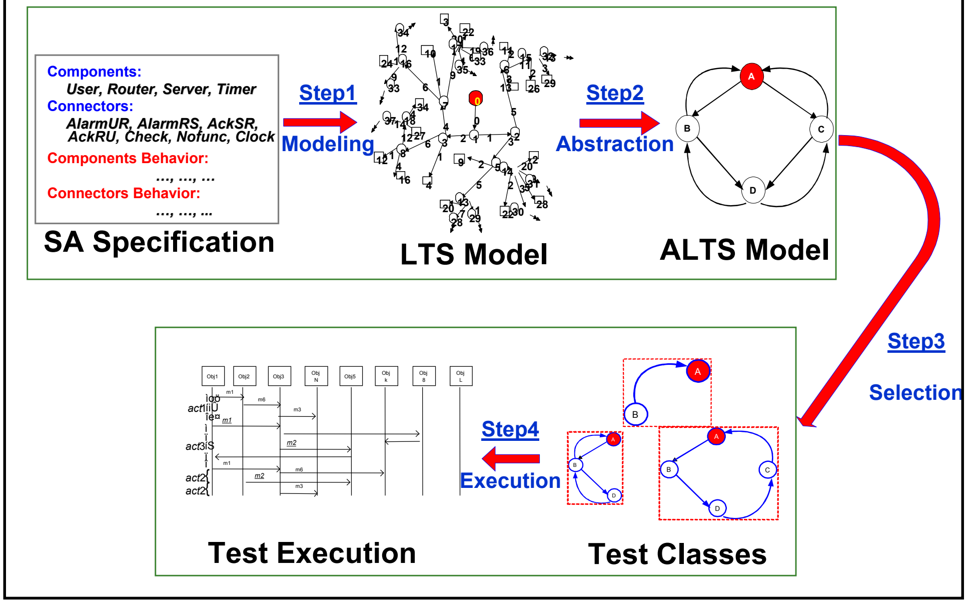
- There exists no such thing as an ideal test plan to accomplish this goal.
- On the contrary, from the high-level, architectural description of a system, several SA-based test plans could be derived, each one addressing the validation of a specific aspect of the system, and of different interaction schemes between components.
- We will describe an approach by which interesting behaviors (i.e., with respect to the testing objectives) can be selected on the global LTS model and can be used to drive code testing.



Testing IUT conformance to SA: Key issues

- *Given an architectural path to be tested, does the system as-built implement this architectural behavior?* E.g., is the sequence of actions "Act1.Act2. …Actn" correctly implemented in the code?
- We have to:
  - identify the set of SA behaviors (=LTS paths) we want to test to establish adequate confidence in conformance ➡ TEST SELECTION
  - understand how the LTS paths are implemented in the IUT ➡ TEST EXECUTION
- *Where do FORMAL METHODS make a difference?*

The Approach



Components:
User, Router, Server, Timer
Connectors:
AlarmUR, AlarmRS, AckSR, AckRU, Check, Nofunc, Clock
Components Behavior:
…, …, …
Connectors Behavior:
…, …, …

SA Specification

Step1
Modeling

LTS Model

Step2
Abstraction

ALTS Model

Step3
Selection

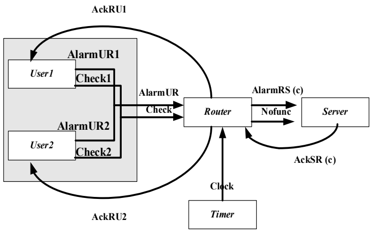
Test Classes

Step4
Execution

Test Execution

Starting point: Formal SA specification

- Any Architectural Description Language (ADL) that produces a Labeled Transition System (LTS), e.g.:
  - **Chemical Abstract Machine (CHAM) [ICSE00]**
  - **Finite State Process (FSP) [ICSE01]**



Molecule syntax:
Molecule: Process| Operation| M.M
Process: User, Router, Server, Timer
Channel : check, alarmUR, alarmRS, ackRU, ackSR, nofunc, clock
Operation: i(Channel), o(Channel), ...

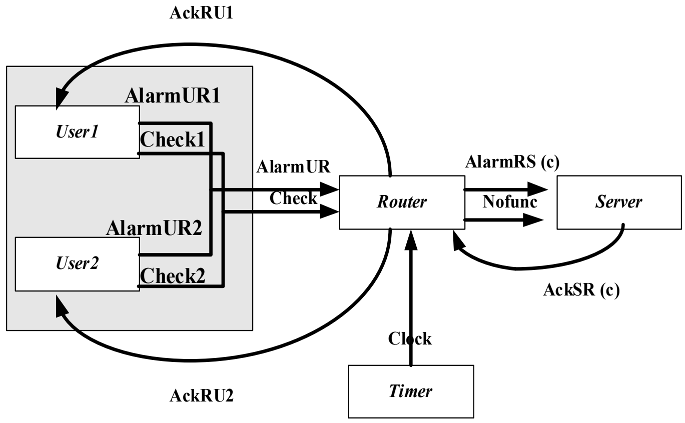
InitialState (S0):
Multiset of Molecules: m1, m2, ..., mn

Rules:
m1, m2, ..., mk →m1', m2', ..., ml'

- Dynamics in terms of Component interactions

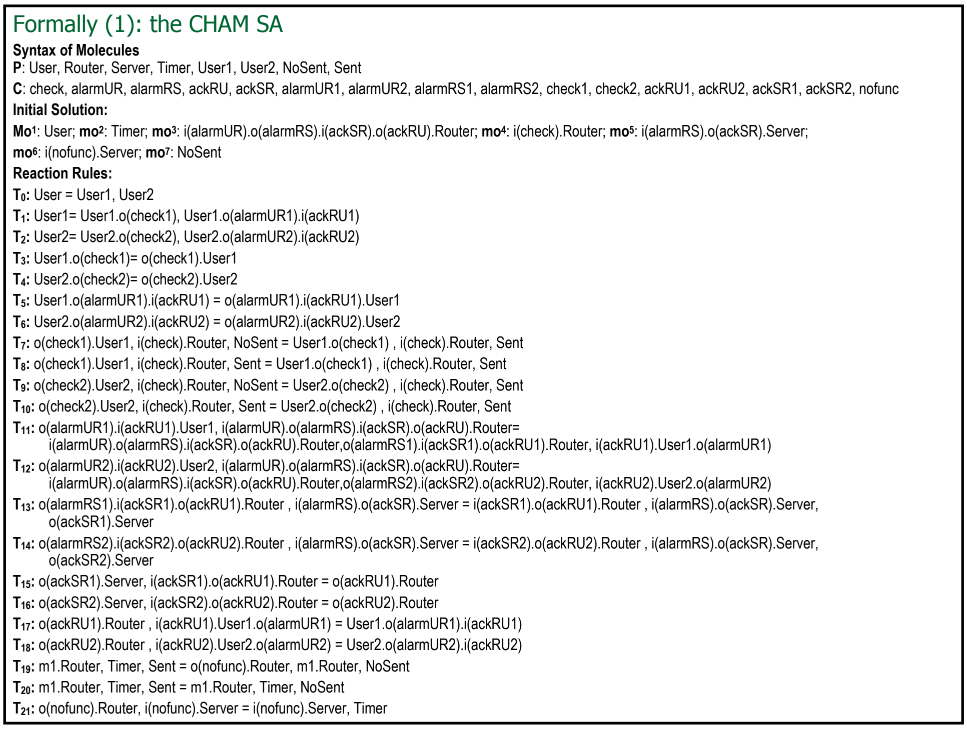
Example: the TRMCS SA

- The Teleservices and Remote Medical Care System provides monitoring and assistance to users with specific needs, like disabled or elderly people.



TRMCS topology

- Four component types:
  - **User:** sends either an “alarm” or a “check” message to the Router process. After sending an alarm, it waits for an ack from the Router.
    - We analyze the case in which two Users can concurrently send alarms and checks.
  - **Router:** waits for signals (check or alarm) from User. It forwards the alarm messages to the Server and monitors the state of the User via check messages.
  - **Server:** dispatches the help requests.
  - **Timer:** sends a clock signal at each time unit.



Formally (1): the CHAM SA

Syntax of Molecules

P: User, Router, Server, Timer, User1, User2, NoSent, Sent

C: check, alarmUR, alarmRS, ackRU, ackSR, alarmUR1, alarmUR2, alarmRS1, alarmRS2, check1, check2, ackRU1, ackRU2, ackSR1, ackSR2, nofunc

Initial Solution:

Mo^1: User; **mo^2**: Timer; **mo^3**: i(alarmUR).o(alarmRS).i(ackSR).o(ackRU).Router; **mo^4**: i(check).Router; **mo^5**: i(alarmRS).o(ackSR).Server;
mo^6: i(nofunc).Server; **mo^7**: NoSent

Reaction Rules:

T_0: User = User1, User2

T_1: User1= User1.o(check1), User1.o(alarmUR1).i(ackRU1)

T_2: User2= User2.o(check2), User2.o(alarmUR2).i(ackRU2)

T_3: User1.o(check1)= o(check1).User1

T_4: User2.o(check2)= o(check2).User2

T_5: User1.o(alarmUR1).i(ackRU1) = o(alarmUR1).i(ackRU1).User1

T_6: User2.o(alarmUR2).i(ackRU2) = o(alarmUR2).i(ackRU2).User2

T_7: o(check1).User1, i(check).Router, NoSent = User1.o(check1) , i(check).Router, Sent

T_8: o(check1).User1, i(check).Router, Sent = User1.o(check1) , i(check).Router, Sent

T_9: o(check2).User2, i(check).Router, NoSent = User2.o(check2) , i(check).Router, Sent

T_{10}: o(check2).User2, i(check).Router, Sent = User2.o(check2) , i(check).Router, Sent

T_{11}: o(alarmUR1).i(ackRU1).User1, i(alarmUR).o(alarmRS).i(ackSR).o(ackRU).Router=
i(alarmUR).o(alarmRS).i(ackSR).o(ackRU).Router,o(alarmRS1).i(ackSR1).o(ackRU1).Router, i(ackRU1).User1.o(alarmUR1)

T_{12}: o(alarmUR2).i(ackRU2).User2, i(alarmUR).o(alarmRS).i(ackSR).o(ackRU).Router=
i(alarmUR).o(alarmRS).i(ackSR).o(ackRU).Router,o(alarmRS2).i(ackSR2).o(ackRU2).Router, i(ackRU2).User2.o(alarmUR2)

T_{13}: o(alarmRS1).i(ackSR1).o(ackRU1).Router , i(alarmRS).o(ackSR).Server = i(ackSR1).o(ackRU1).Router , i(alarmRS).o(ackSR).Server,
o(ackSR1).Server

T_{14}: o(alarmRS2).i(ackSR2).o(ackRU2).Router , i(alarmRS).o(ackSR).Server = i(ackSR2).o(ackRU2).Router , i(alarmRS).o(ackSR).Server,
o(ackSR2).Server

T_{15}: o(ackSR1).Server, i(ackSR1).o(ackRU1).Router = o(ackRU1).Router

T_{16}: o(ackSR2).Server, i(ackSR2).o(ackRU2).Router = o(ackRU2).Router

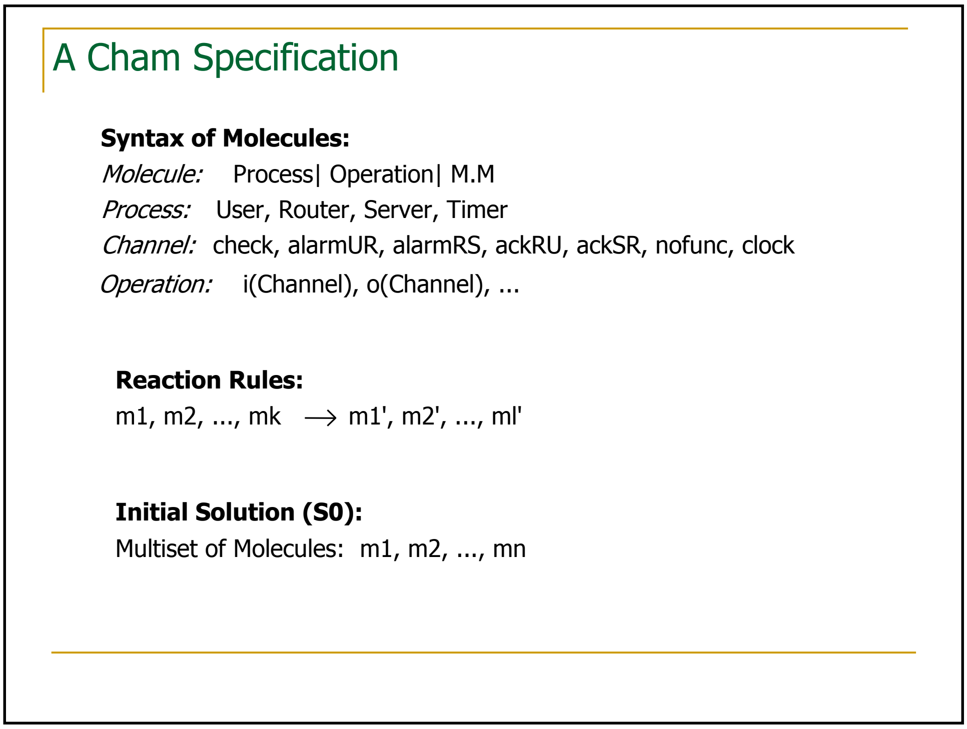
T_{17}: o(ackRU1).Router , i(ackRU1).User1.o(alarmUR1) = User1.o(alarmUR1).i(ackRU1)

T_{18}: o(ackRU2).Router , i(ackRU2).User2.o(alarmUR2) = User2.o(alarmUR2).i(ackRU2)

T_{19}: m1.Router, Timer, Sent = o(nofunc).Router, m1.Router, NoSent

T_{20}: m1.Router, Timer, Sent = m1.Router, Timer, NoSent

T_{21}: o(nofunc).Router, i(nofunc).Server = i(nofunc).Server, Timer



A Cham Specification

Syntax of Molecules:

Molecule: Process| Operation| M.M

Process: User, Router, Server, Timer

Channel: check, alarmUR, alarmRS, ackRU, ackSR, nofunc, clock

Operation: i(Channel), o(Channel), ...

Reaction Rules:

m1, m2, ..., mk $\longrightarrow$ m1', m2', ..., ml'

Initial Solution (S0):

Multiset of Molecules: m1, m2, ..., mn

Formally (2): the FSP SA

```
range N = 0..1
range K = 0..1
range Sent = 0..1
```

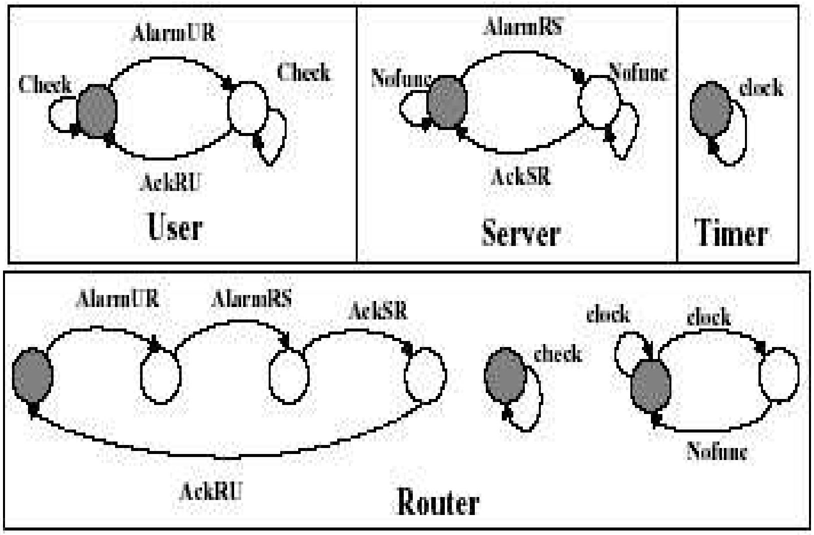
```
/** User Process   */
USER_ALARM= (sendAlarm_To_Router -> receiveAck_From_Router -> USER_ALARM).

USER_CHECK = USER_SENDCHECK, USER_SENDCHECK = (sendCheck_To_Router ->USER_SENDCHECK).

||USER = (USER_ALARM||USER_CHECK).
```

```
/** Router Process */

ROUTER_RECEIVEALARM = (receiveAlarm_From_User -> sendAlarm_To_Server ->
receiveAck_From_Server -> sendAck_To_User -> ROUTER_RECEIVEALARM).


ROUTER_RECEIVECHECK = (receiveCheck_From_User -> (sendInput_To_Timer ->
ROUTER_RECEIVECHECK|pre_receiveCheck -> ROUTER_RECEIVECHECK)).


ROUTER_RECEIVETIME = (receiveTime_From_Timer ->
(sendNoFunc_To_Server -> ROUTER_RECEIVETIME|pre_receiveTime-> ROUTER_RECEIVETIME)).


||ROUTER =([0..1]:ROUTER_RECEIVEALARM||[0..1]:ROUTER_RECEIVECHECK||ROUTER_RECEIVETIME).
```

```
/** Server Process */

SERVER_RECEIVEALARM = (receiveAlarm_From_Router -> sendAck_To_Router -> SERVER_RECEIVEALARM).

SERVER_RECEIVETIME = (receiveNoFunc_From_Router -> SERVER_RECEIVETIME).
```

```
/** Timer Process */

TIMER = (receiveInput_From_Router -> sendTime_To_Router -> TIMER).
```

TRMCS SA behavior

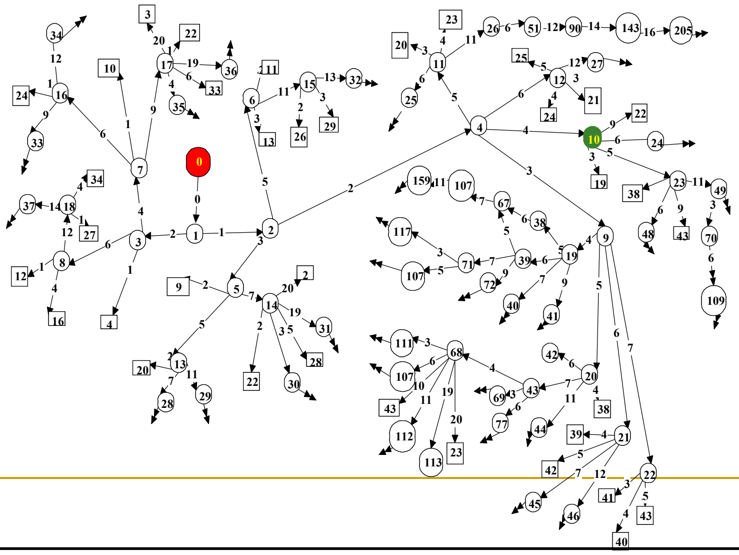


Formally (2): the FSP SA

```
▪  /** User_Router_Server_Timer */
 ||USER_ROUTER=(u[0..1]:USER||r:ROUTER||sa[0..1]:SERVER_RECEIVEALARM||st:SERVER_REC
   EIVETIME||t:TIMER)/
{  u[0].sendAlarm_To_Router/r.[0].receiveAlarm_From_User,
   u[1].sendAlarm_To_Router/r.[1].receiveAlarm_From_User,

   r.[0].sendAlarm_To_Server/sa[0].receiveAlarm_From_Router,
   r.[1].sendAlarm_To_Server/sa[1].receiveAlarm_From_Router,

   sa[0].sendAck_To_Router/r.[0].receiveAck_From_Server,
   sa[1].sendAck_To_Router/r.[1].receiveAck_From_Server,

   r.[0].sendAck_To_User/u[0].receiveAck_From_Router,
   r.[1].sendAck_To_User/u[1].receiveAck_From_Router,

   u[0].sendCheck_To_Router/r.[0].receiveCheck_From_User,
   u[1].sendCheck_To_Router/r.[1].receiveCheck_From_User,

   r.[0].sendInput_To_Timer/t.receiveInput_From_Router,
   r.[1].sendInput_To_Timer/t.receiveInput_From_Router,

   t.sendTime_To_Router/r.receiveTime_From_Timer,

   r.sendNoFunc_To_Server/st.receiveNoFunc_From_Router
 }.
```

From any formal SA, we need to get to the LTS

in order to obtain a rigorous description of the Architecture-level Behavior



We could now automatically derive the SA tests by applying some notion of coverage on the LTS, but:

- The LTS provides a monolithic description of SA behavior; it is a vast amount of information flattened into a graph (lack of intuitive grasp)
- How can we extract from this global model those observations (samples) of SA behavior that are relevant for testing?

The idea:

We apply a "*SA testing criterion*": formally defined in terms of an

obs-function on the LTS

Step 1: definition of the *obs*-function

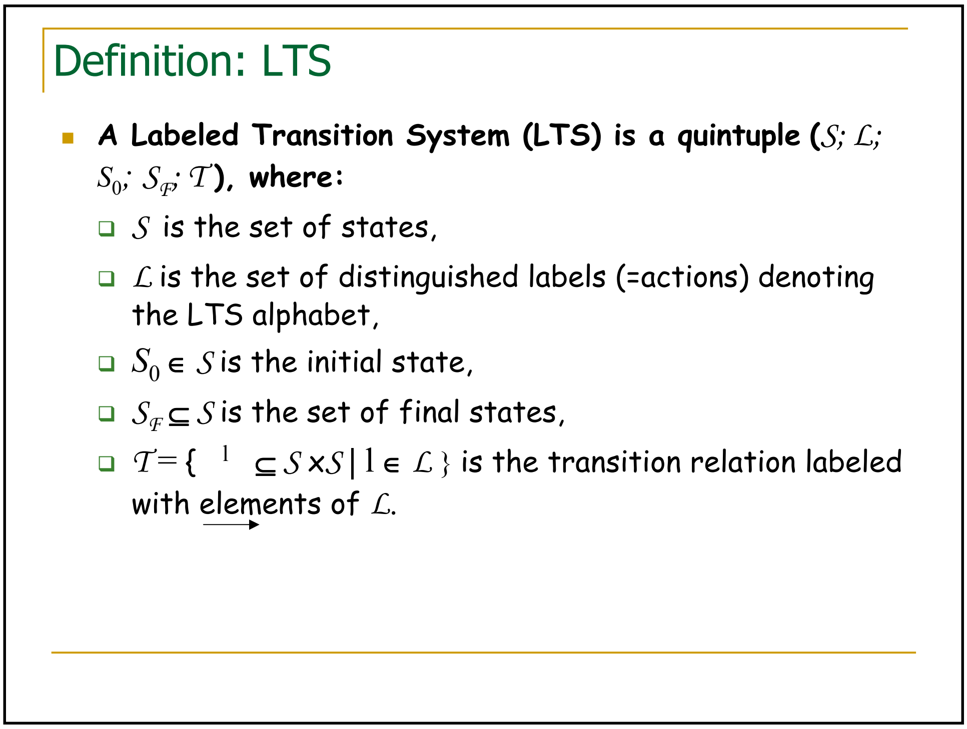
- Intuition Behind:

Architectural Views

- Flow view
- Component Based view
- Concurrency view
- …

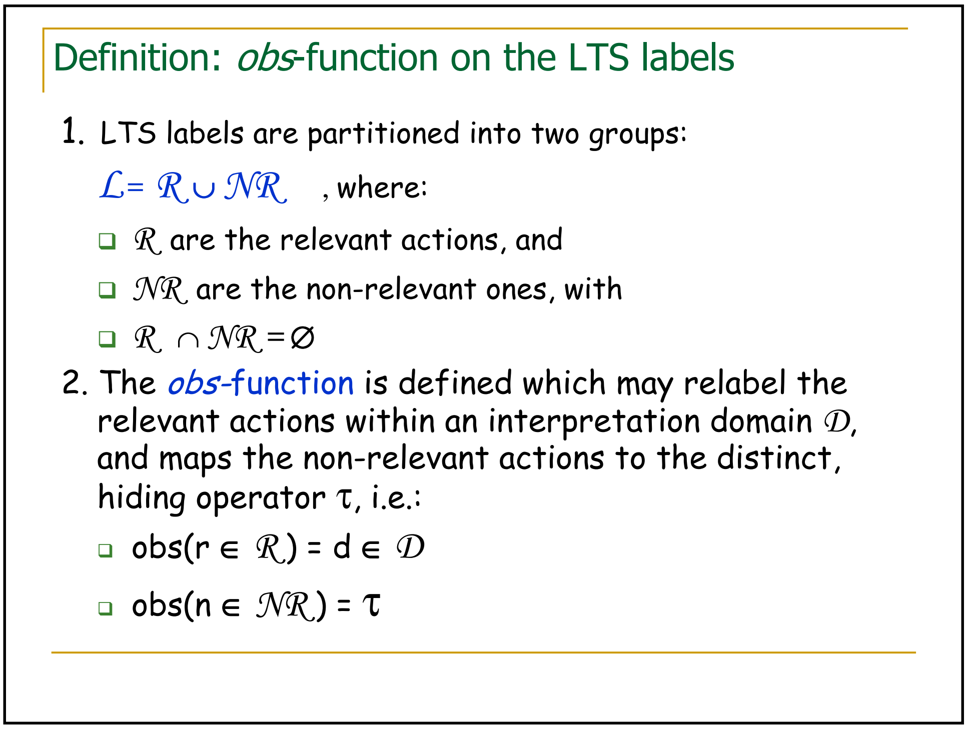
Or, a Criterion to decide "equivalent" tests ("test class").

- Obs_Functions look at the SA each from a perspective that is deemed to be relevant for testing purposes;
- non-relevant actions from this perspective are hidden away.



Definition: LTS

- **A Labeled Transition System (LTS) is a quintuple (**$\mathcal{S}; \mathcal{L}; \mathcal{S}_0; \mathcal{S}_{\mathcal{F}}; \mathcal{T}$**), where:**
  - $\mathcal{S}$ is the set of states,
  - $\mathcal{L}$ is the set of distinguished labels (=actions) denoting the LTS alphabet,
  - $\mathcal{S}_0 \in \mathcal{S}$ is the initial state,
  - $\mathcal{S}_{\mathcal{F}} \subseteq \mathcal{S}$ is the set of final states,
  - $\mathcal{T} = \{ \xrightarrow{l} \subseteq \mathcal{S} \times \mathcal{S} \mid l \in \mathcal{L} \}$ is the transition relation labeled with elements of $\mathcal{L}$.



Definition: *obs*-function on the LTS labels

1. LTS labels are partitioned into two groups:

   $\mathcal{L} = \mathcal{R} \cup \mathcal{NR}$, where:
   - $\mathcal{R}$ are the relevant actions, and
   - $\mathcal{NR}$ are the non-relevant ones, with
   - $\mathcal{R} \cap \mathcal{NR} = \varnothing$
2. The *obs*-function is defined which may relabel the relevant actions within an interpretation domain $\mathcal{D}$, and maps the non-relevant actions to the distinct, hiding operator τ, i.e.:
   - obs(r ∈ $\mathcal{R}$) = d ∈ $\mathcal{D}$
   - obs(n ∈ $\mathcal{NR}$) = τ

Example: AlarmObs

- Intuitively ("test criterion"):
  all those behaviors involving the flow of an Alarm message through the system

- Formally (*obs*-function):

$\mathcal{D}$ = {SendAlarm1, SendAlarm2, ReceiveAck1, ReceiveAck2} , *with*

obs (u.0.sendAlarm_To_Router) =SendAlarm1
User1 issues an Alarm msg

obs (u.0.receiveAck_From_Router) = ReceiveAck1
User1 receives an Ack

obs(u.1.sendAlarm_To_Router) = SendAlarm2
User2 issues an Alarm msg

obs (u.1.receiveAck_From_Router) = ReceiveAck2
User2 receives an Ack

and

For any other action r, obs (r) = τ

Another example: ServerObs

- Intuitively ("test criterion"):
  all the interactions involving a specified component (the server)
- Formally (*obs*-function):

$\mathcal{D}$ = {FRa1, FRa2, TRack1, TRack2, FRno} , *with*

obs (r.[0].sendAlarm_To_Server)= FRa1: *Alarm1 From Router to Server*
obs(r.[1].sendAlarm_To_Server)= FRa2: *Alarm2 From Router to Server*
obs(sa[0].sendAck_To_Router) = TRack1: *Ack1 From Server to Router*
obs (sa[1].sendAck_To_Router) = TRack2: *Ack2 From Server to Router*
obs (r.sendNoFunc_To_Server) = FRno: *NoFunctioning From Router to Server*

and

For any other action r, obs (r) = τ

Step 2 : The Abstract LTS

- We use the *obs*-function as a means to derive a smaller automaton from the LTS:
  the Abstract LTS or ALTS

- The ALTS is obtained via two transformations:

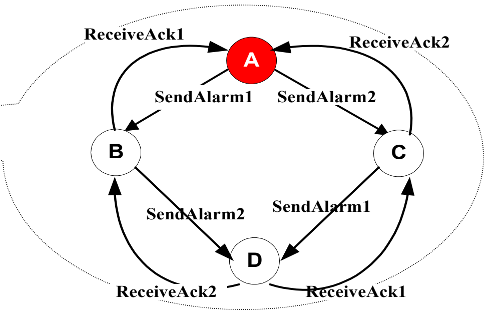
1. Graph relabelling according to the *obs*-function (we obtain the ObsLTS)

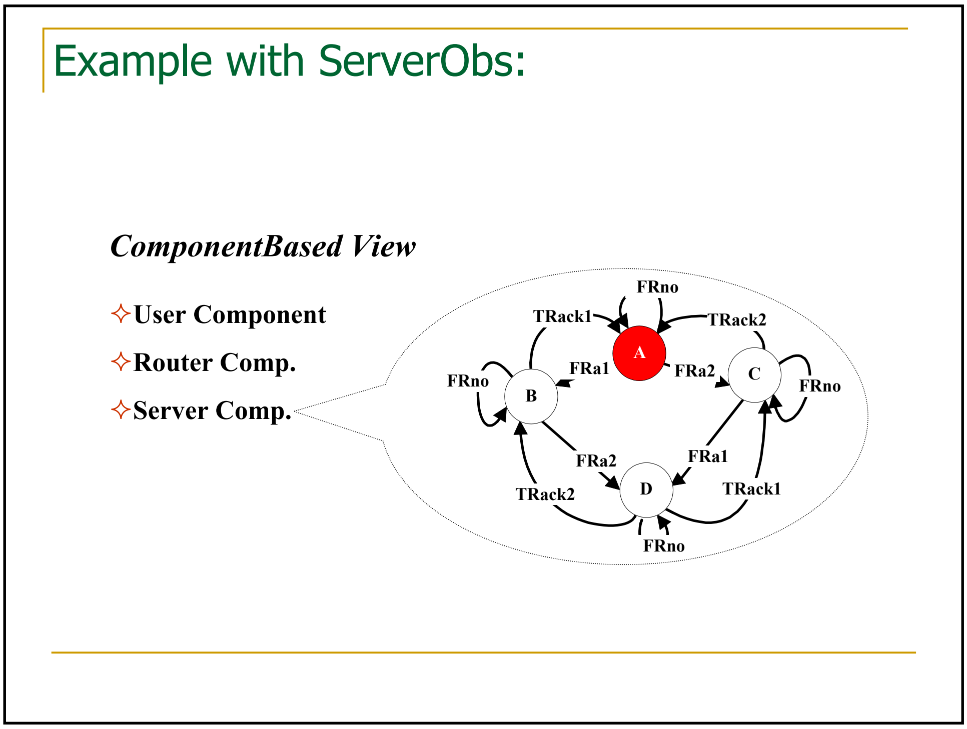
2. Minimization (trace-equivalence)

Example with AlarmObs:

Flow view:

- ✧**Alarm Flow**
- ✧**Check Flow**
- ✧**Clock Flow**

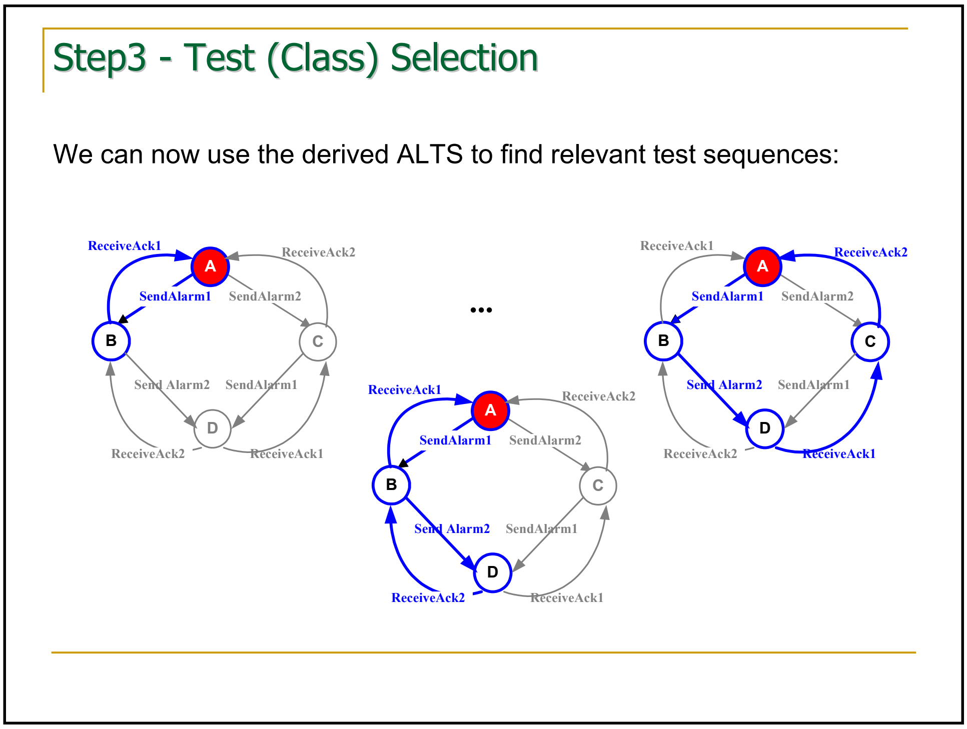




Example with ServerObs:

ComponentBased View

- ✧User Component
- ✧Router Comp.
- ✧Server Comp.

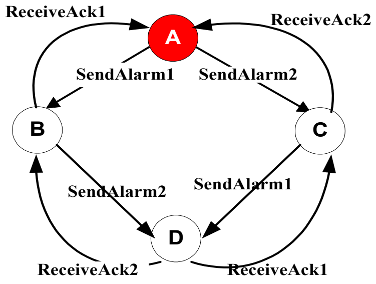


Step3 - Test (Class) Selection

We can now use the derived ALTS to find relevant test sequences:

Coverage Criterion *over the ALTS*

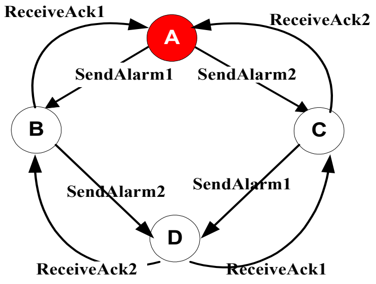
- SA-based test selection is thus reduced to finding an "appropriate" set of paths covering the ALTS



- Each complete path (entry-exit) corresponds to an Architectural Test Class

ALTS Coverage: *how?*

- *Trade-off between exhaustiveness and cost:*
  a path on the ALTS will then be refined in many concrete tests



We rather adopt a loose criterion: for instance, all the indipendent paths (McCabe's cyclomatic complexity):

CC(G) = #edges - #nodes + 1 -> 8-4+1=5

P1= A B A

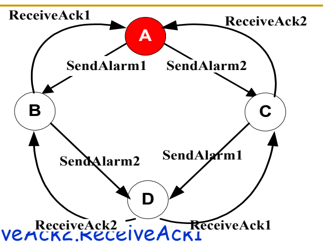
P2= A B D B A

P3= A B A C A

P4= A B D C A

P5= A B D C D C A

Intuition



P1= A B A → SendAlarm1.ReceiveAck1

Basic functioning of msg exchange

P2= A B D B A → SendAlarm1. SendAlarm2.ReceiveAck2.ReceiveAck1

P3= A B A C A → SendAlarm1 .ReceiveAck1. SendAlarm2.ReceiveAck2

P4= A B D C A → SendAlarm1. SendAlarm2. ReceiveAck1.ReceiveAck2

The system correctly handles the consecutive reception of two Alarm msgs issued by two distinct Users

----

P5= A B D C D C A → SendAlarm1. SendAlarm2.ReceiveAck1. SendAlarm1. ReceiveAck1.ReceiveAck2

More complex interleaving of Alarm msgs issued by two distinct Users

- NB: Any other path can be obtained by combining the independent paths

ALTS vs. LTS

ALTS is a tool to select test sequences.

After an ALTS path is chosen, we may want to go back to the complete LTS and recover the information that was previously hidden (to identify SA meaningful test sequences).

Conformance testing from ???

- ALTS paths: more abstract, focussed, and intuitive sequences
- LTS paths: more closely adhering to the SA models

  Differing levels of abstraction and contexts

From ALTS back to LTS …

Given a generic ALTS path:

$P = A \xrightarrow{l1} B \xrightarrow{l2} C$

We can

1. Apply the inverse of *obs*-function, e.g.:

   $Obs^{-1}(l1)=a1$, $Obs^{-1}(l2)=a2$,

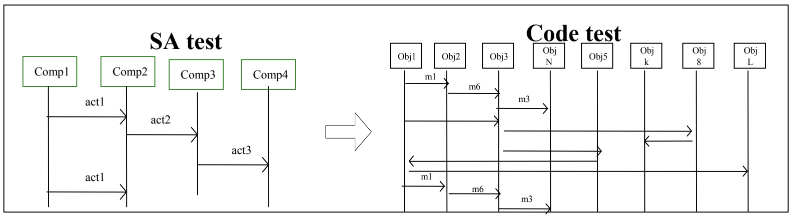
2. We look on the LTS for any paths such that:

   P'= p1.a1.p2.a2.p3, where p1, p2 and p3 are (possibly empty) subpaths only containing LTS actions mapped to τ (*graph manipulation algorithms*)

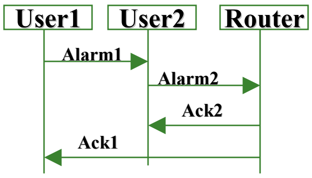
Clearly, several LTS paths will correspond to each ALTS (again find selection criteria)

Step4 - Testing the Software Implementation

- How are the ALTS/LTS paths implemented by the Code?
  - **We want to test whether the system as-built implements the selected architectural behavior**



Different development processes



SA behavior

??? ⟷

```
class MasterRouter{
   ServerConnection allarm;
   ServerConnection okFunction;
   // the services
   ReceiveUserAllarm serviceReceiveUserAllarm;
   ReceiveUserOkFunz serviceReceiveUserOkFunz;
   String name,serverName;
   static public PrintWriter user_router_ok_funz;
   static public PrintWriter user_router_alarm;
   ...
```

Source Code

SA (topology and model) *drives* → **Design and Source Code Def.**

SA (model) *drives* ← **Source Code** (abstractions)

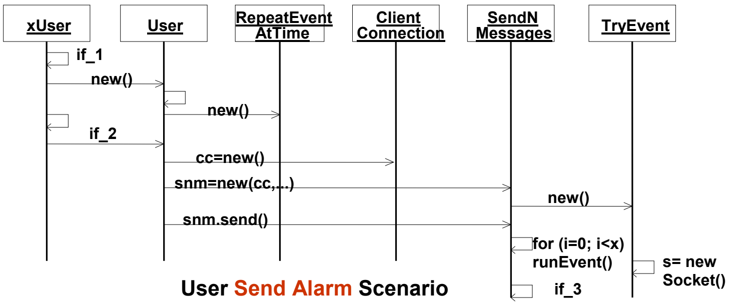
SA (model) ⟷̸ **Source Code**

From SA tests to code tests

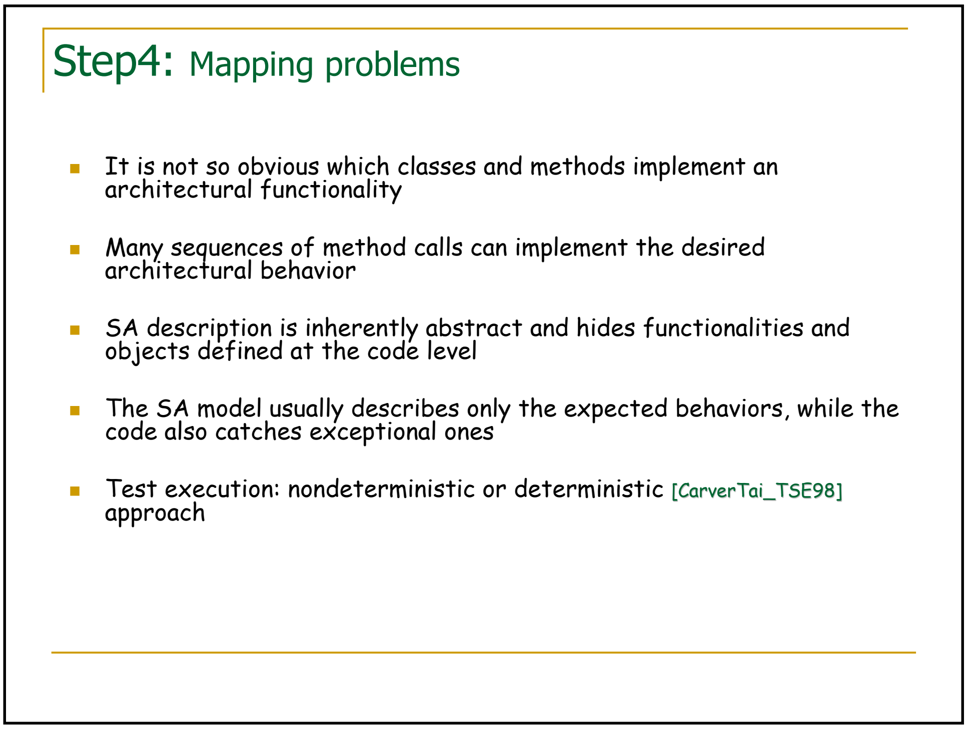
Given the sequence of actions to be tested: Act1.Act2…Actn:

… a test class is an ordered sequence of actions…
…each action is implemented by a sequence of methods…
… then an architectural sequence is implemented by the sequencial (or concurrent) execution of different sequences of methods.

For each architectural action, we have to identify the set of method calls implementing it :

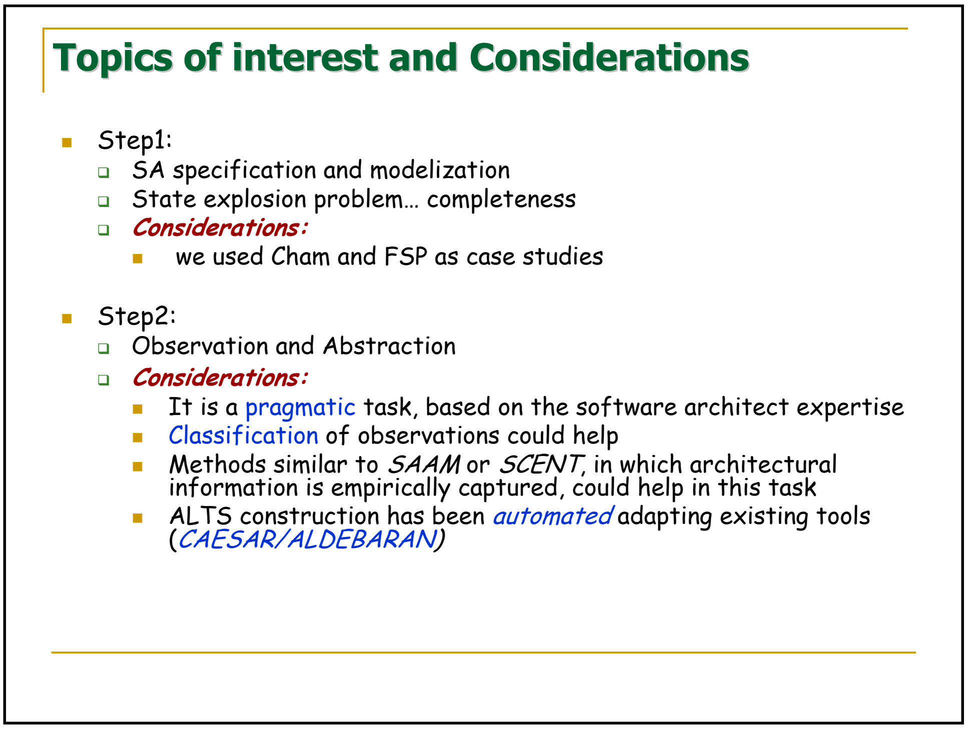


User Send Alarm Scenario



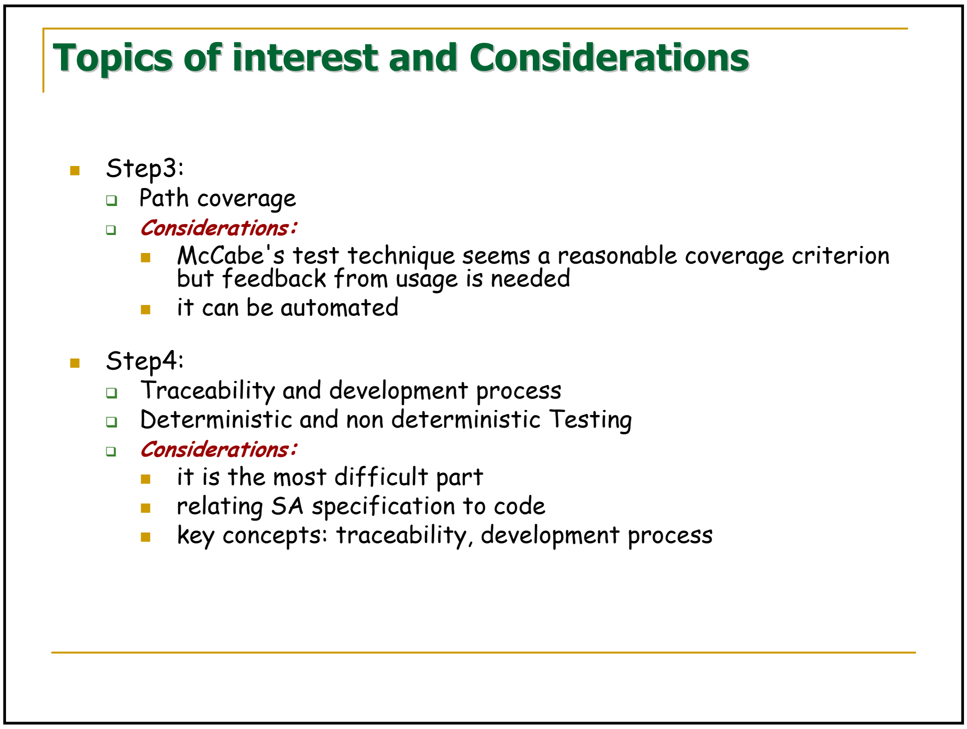
Step4: Mapping problems

- It is not so obvious which classes and methods implement an architectural functionality
- Many sequences of method calls can implement the desired architectural behavior
- SA description is inherently abstract and hides functionalities and objects defined at the code level
- The SA model usually describes only the expected behaviors, while the code also catches exceptional ones
- Test execution: nondeterministic or deterministic [CarverTai_TSE98] approach



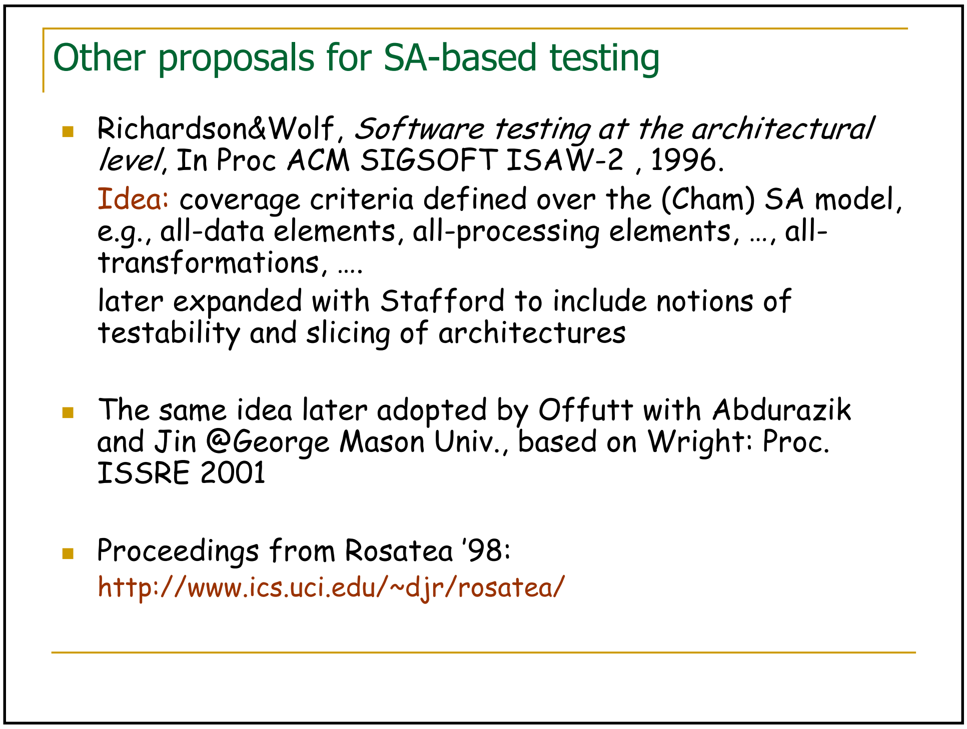
Topics of interest and Considerations

- Step1:
  - SA specification and modelization
  - State explosion problem… completeness
  - ***Considerations:***
    - we used Cham and FSP as case studies
- Step2:
  - Observation and Abstraction
  - ***Considerations:***
    - It is a pragmatic task, based on the software architect expertise
    - Classification of observations could help
    - Methods similar to *SAAM* or *SCENT*, in which architectural information is empirically captured, could help in this task
    - ALTS construction has been *automated* adapting existing tools (*CAESAR/ALDEBARAN)*



Topics of interest and Considerations

- Step3:
  - Path coverage
  - *Considerations:*
    - McCabe's test technique seems a reasonable coverage criterion but feedback from usage is needed
    - it can be automated
- Step4:
  - Traceability and development process
  - Deterministic and non deterministic Testing
  - *Considerations:*
    - it is the most difficult part
    - relating SA specification to code
    - key concepts: traceability, development process



Other proposals for SA-based testing

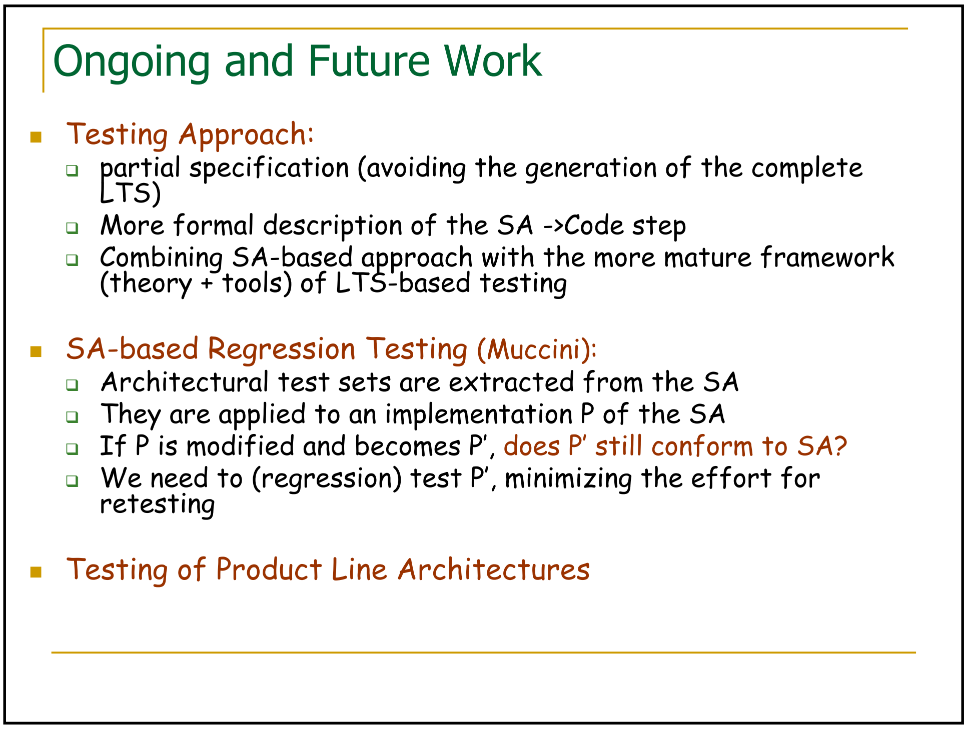
- Richardson&Wolf, *Software testing at the architectural level*, In Proc ACM SIGSOFT ISAW-2 , 1996.

  Idea: coverage criteria defined over the (Cham) SA model, e.g., all-data elements, all-processing elements, …, all-transformations, ….

  later expanded with Stafford to include notions of testability and slicing of architectures

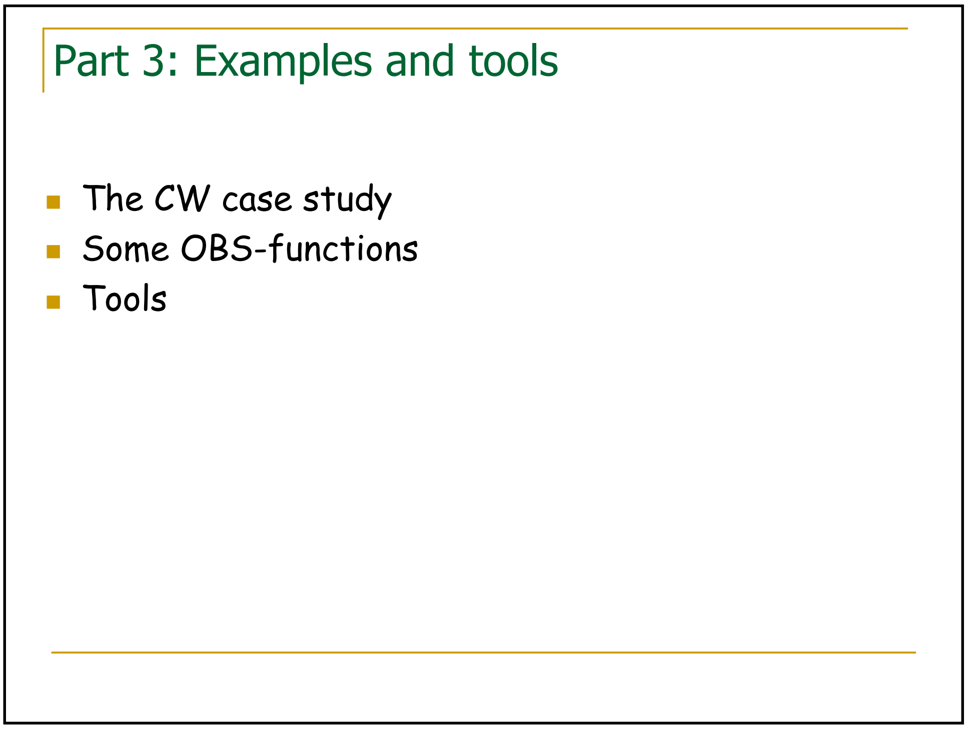
- The same idea later adopted by Offutt with Abdurazik and Jin @George Mason Univ., based on Wright: Proc. ISSRE 2001

- Proceedings from Rosatea '98: http://www.ics.uci.edu/~djr/rosatea/



Ongoing and Future Work

- Testing Approach:
  - partial specification (avoiding the generation of the complete LTS)
  - More formal description of the SA ->Code step
  - Combining SA-based approach with the more mature framework (theory + tools) of LTS-based testing
- SA-based Regression Testing (Muccini):
  - Architectural test sets are extracted from the SA
  - They are applied to an implementation P of the SA
  - If P is modified and becomes P', does P' still conform to SA?
  - We need to (regression) test P', minimizing the effort for retesting
- Testing of Product Line Architectures



Part 3: Examples and tools

- The CW case study
- Some OBS-functions
- Tools

Collaborative Writing:

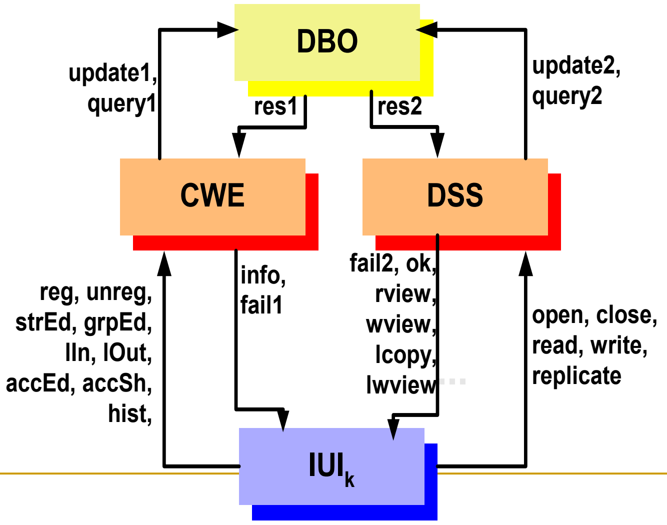
the process by which authors with different expertise and responsibilities interact during the creation and revision of a common document. Concurrency control policies (either pessimistic or optimistic) are adopted to manage concurrent access to a same doc.

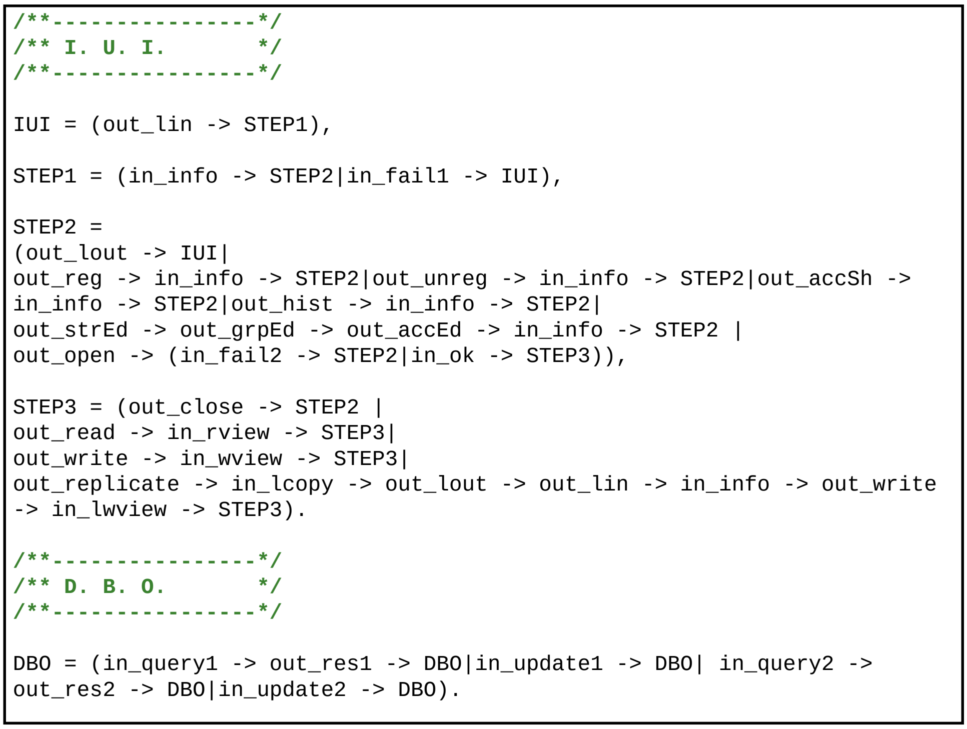
Four Components:

- Integrated User Interface (IUI): an integrated environment of tools for editing, navigation, display of awareness communication and import/export of data from external applications;
- CW Engine (CWE): all services useful to perform group activities, e.g., user registration and deregistration, user login and logout, users group definition and editing;
- Document Storage Service (DSS): abstract container of shared documents structured in partitions. In *asynchronous* mode, version-controlled documents; in *synchronous* mode, a pessimistic concurrency control;
- Database Object (DBO): stores all group awareness information useful to support group activities.

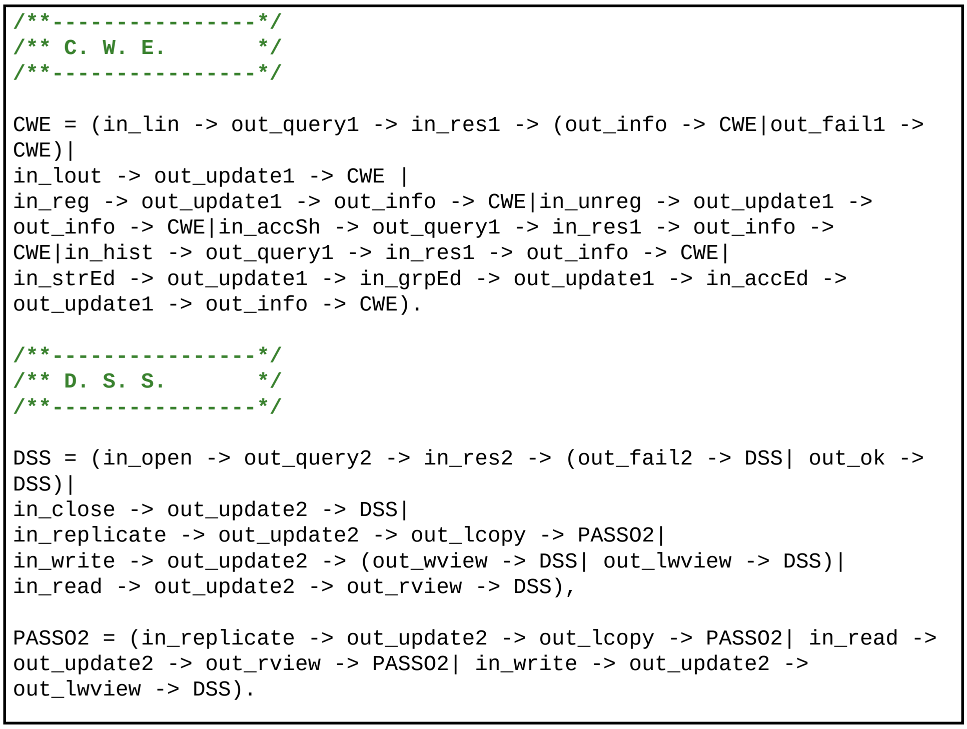
CW SA

depicts the different components, the architecture topology and the list of services each component provides and/or requires (www.di.univaq.it/tivoli/cscv_techrep.pdf)

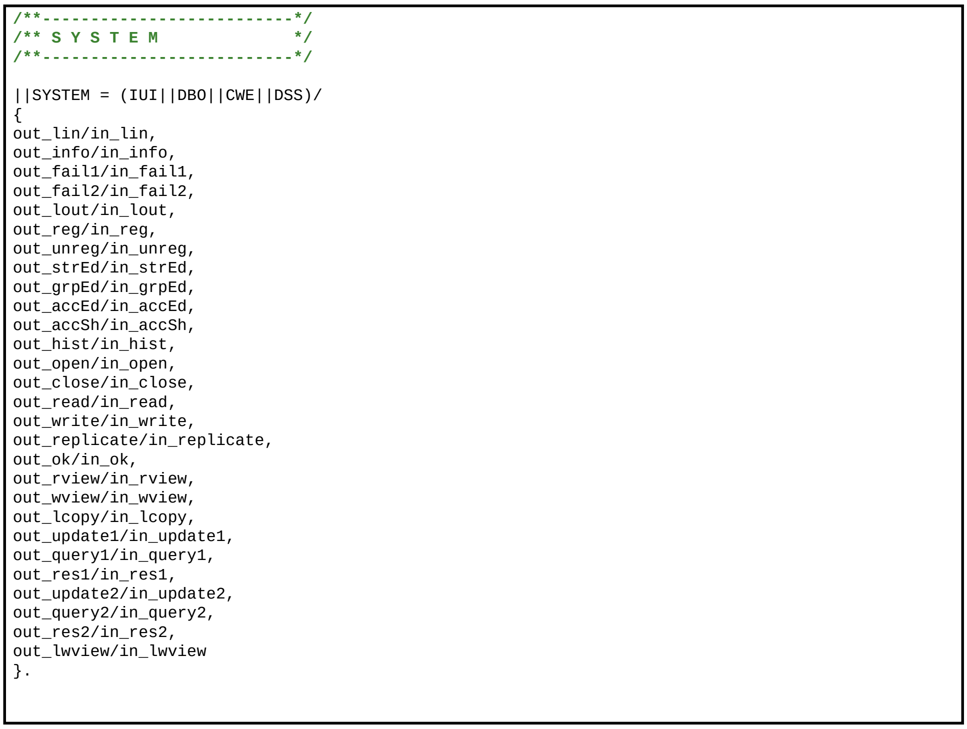

DBO
update1, query1
res1
res2
update2, query2
CWE
DSS
reg, unreg, strEd, grpEd, lIn, lOut, accEd, accSh, hist,
info, fail1
fail2, ok, rview, wview, lcopy, lwview…
open, close, read, write, replicate
IUI$_k$



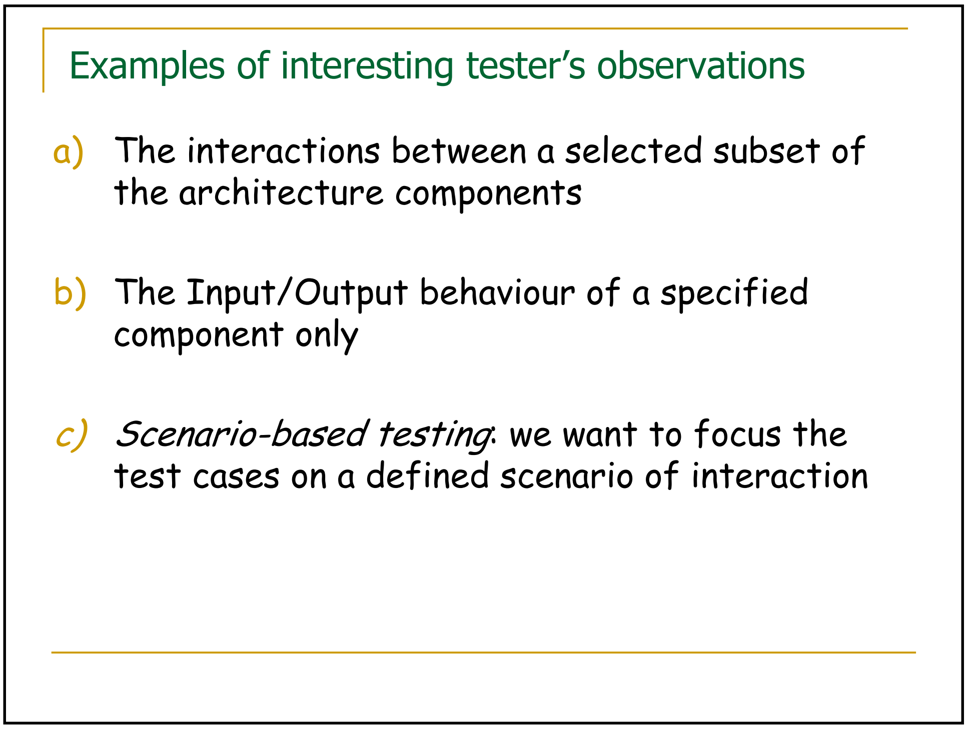
```
/**----------------*/
/** I. U. I.       */
/**----------------*/

IUI = (out_lin -> STEP1),

STEP1 = (in_info -> STEP2|in_fail1 -> IUI),

STEP2 =
(out_lout -> IUI|
out_reg -> in_info -> STEP2|out_unreg -> in_info -> STEP2|out_accSh ->
in_info -> STEP2|out_hist -> in_info -> STEP2|
out_strEd -> out_grpEd -> out_accEd -> in_info -> STEP2 |
out_open -> (in_fail2 -> STEP2|in_ok -> STEP3)),

STEP3 = (out_close -> STEP2 |
out_read -> in_rview -> STEP3|
out_write -> in_wview -> STEP3|
out_replicate -> in_lcopy -> out_lout -> out_lin -> in_info -> out_write
-> in_lwview -> STEP3).

/**----------------*/
/** D. B. O.       */
/**----------------*/

DBO = (in_query1 -> out_res1 -> DBO|in_update1 -> DBO| in_query2 ->
out_res2 -> DBO|in_update2 -> DBO).
```



```
/**----------------*/
/** C. W. E.       */
/**----------------*/

CWE = (in_lin -> out_query1 -> in_res1 -> (out_info -> CWE|out_fail1 ->
CWE)|
in_lout -> out_update1 -> CWE |
in_reg -> out_update1 -> out_info -> CWE|in_unreg -> out_update1 ->
out_info -> CWE|in_accSh -> out_query1 -> in_res1 -> out_info ->
CWE|in_hist -> out_query1 -> in_res1 -> out_info -> CWE|
in_strEd -> out_update1 -> in_grpEd -> out_update1 -> in_accEd ->
out_update1 -> out_info -> CWE).

/**----------------*/
/** D. S. S.       */
/**----------------*/

DSS = (in_open -> out_query2 -> in_res2 -> (out_fail2 -> DSS| out_ok ->
DSS)|
in_close -> out_update2 -> DSS|
in_replicate -> out_update2 -> out_lcopy -> PASSO2|
in_write -> out_update2 -> (out_wview -> DSS| out_lwview -> DSS)|
in_read -> out_update2 -> out_rview -> DSS),

PASSO2 = (in_replicate -> out_update2 -> out_lcopy -> PASSO2| in_read ->
out_update2 -> out_rview -> PASSO2| in_write -> out_update2 ->
out_lwview -> DSS).
```



```
/**-------------------------*/
/** S Y S T E M             */
/**-------------------------*/

||SYSTEM = (IUI||DBO||CWE||DSS)/
{
out_lin/in_lin,
out_info/in_info,
out_fail1/in_fail1,
out_fail2/in_fail2,
out_lout/in_lout,
out_reg/in_reg,
out_unreg/in_unreg,
out_strEd/in_strEd,
out_grpEd/in_grpEd,
out_accEd/in_accEd,
out_accSh/in_accSh,
out_hist/in_hist,
out_open/in_open,
out_close/in_close,
out_read/in_read,
out_write/in_write,
out_replicate/in_replicate,
out_ok/in_ok,
out_rview/in_rview,
out_wview/in_wview,
out_lcopy/in_lcopy,
out_update1/in_update1,
out_query1/in_query1,
out_res1/in_res1,
out_update2/in_update2,
out_query2/in_query2,
out_res2/in_res2,
out_lwview/in_lwview
}.
```

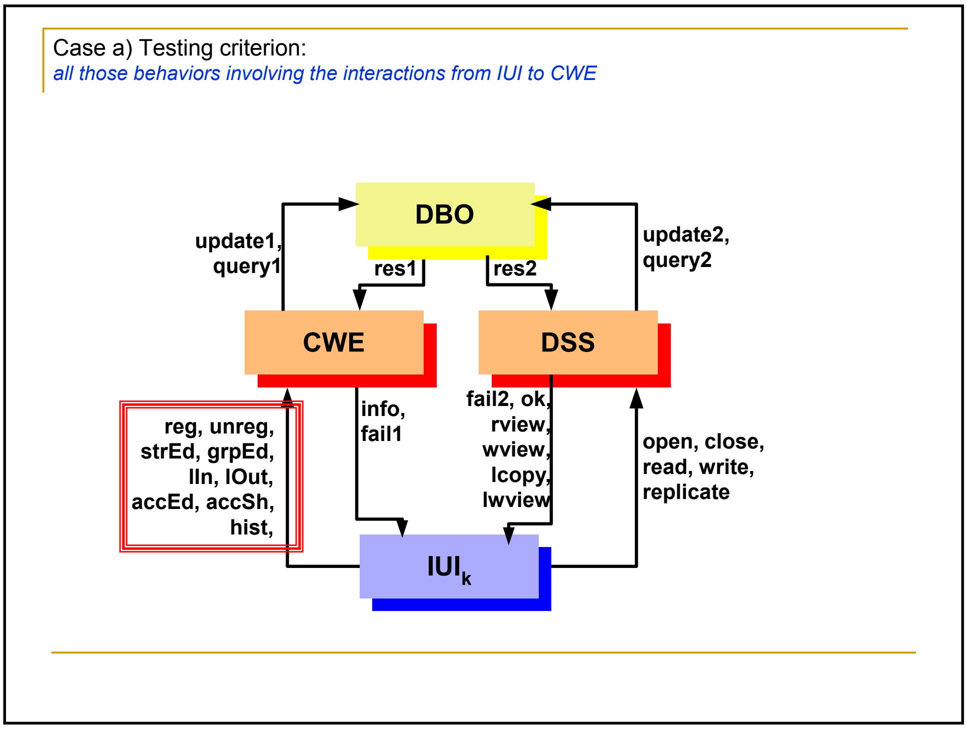


Examples of interesting tester's observations

a) The interactions between a selected subset of the architecture components

b) The Input/Output behaviour of a specified component only

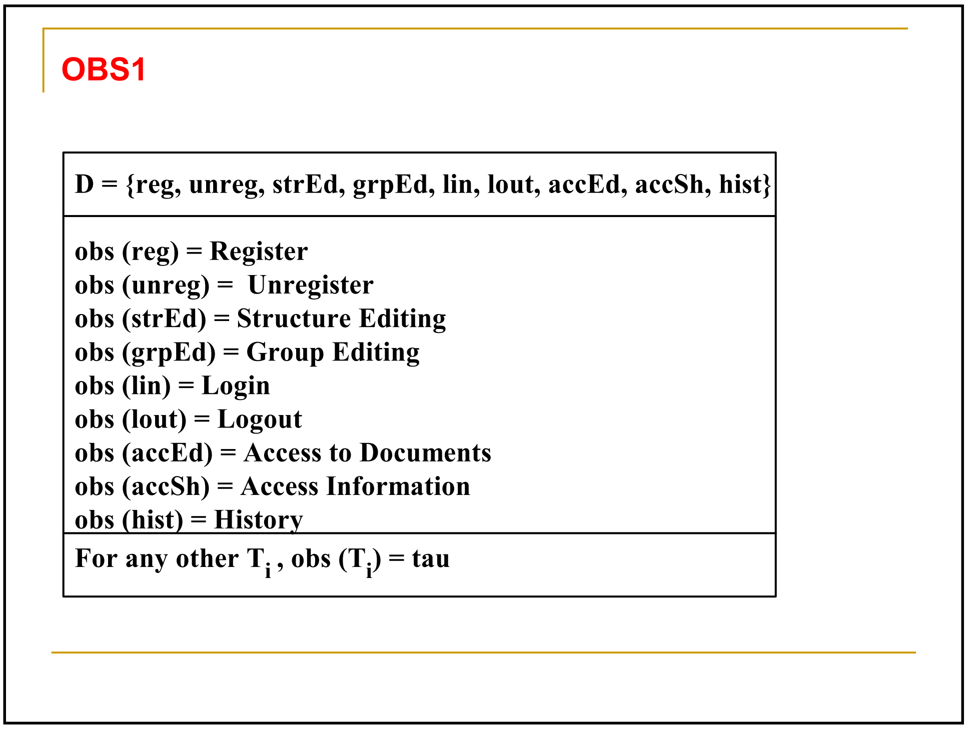
c) *Scenario-based testing*: we want to focus the test cases on a defined scenario of interaction



Case a) Testing criterion:

all those behaviors involving the interactions from IUI to CWE

DBO
CWE
DSS
IUI_k
update1, query1
res1
res2
update2, query2
reg, unreg, strEd, grpEd, lIn, lOut, accEd, accSh, hist,
info, fail1
fail2, ok, rview, wview, lcopy, lwview
open, close, read, write, replicate



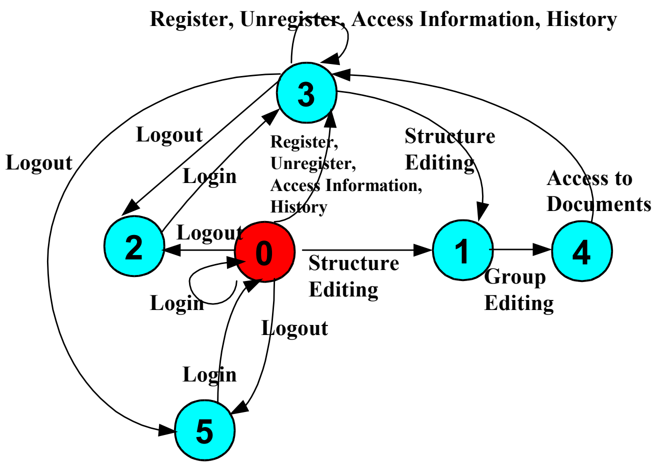
OBS1

D = {reg, unreg, strEd, grpEd, lin, lout, accEd, accSh, hist}

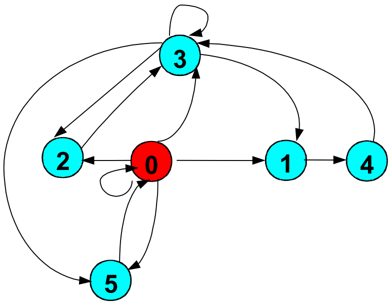
obs (reg) = Register
obs (unreg) = Unregister
obs (strEd) = Structure Editing
obs (grpEd) = Group Editing
obs (lin) = Login
obs (lout) = Logout
obs (accEd) = Access to Documents
obs (accSh) = Access Information
obs (hist) = History

For any other T_i , obs (T_i) = tau

ALTS1



Covering ALTS1



13 arcs, 6 nodes =>
8 independent paths

e.g.,
p= {0} StructureEditing {1} Group Editing {4} AccesstoDocuments {3} Logout {5} Login {0}

The IUI component initially edits the document structure, group information, then accesses the document, and eventually logs out.

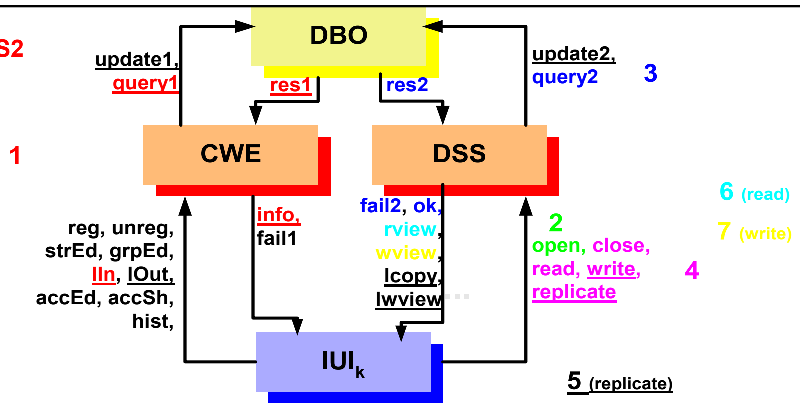
Case c) Testing criterion:

the OPEN scenario, i.e., all the operations related to the execution of the open action

OBS2

obs(pre.open.postOpen) = pre.open.postOpen
obs(pre.z.postZ) = pre.tau, with z different from open

OBS2



A set of scenarios all involving the OPEN action:

1) IUI sends lIn (=login) to CWE, CWE sends query1 to DBO, DBO sends res1 to CWE which sends info to IUI.

2) After these actions, IUI can send OPEN to DSS.

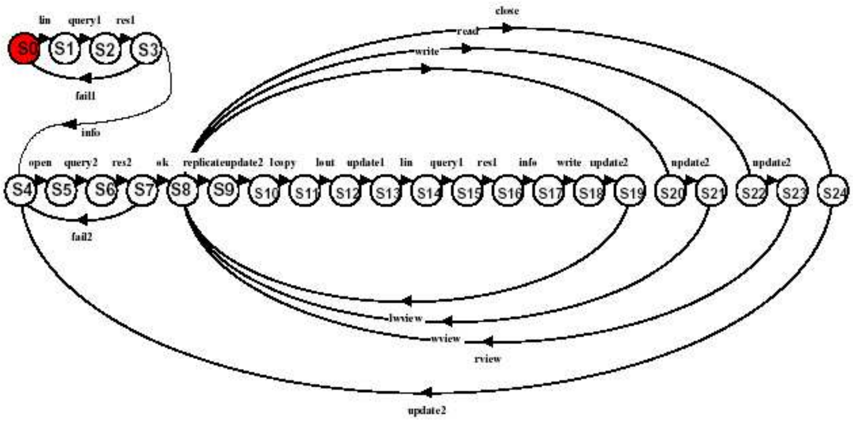
3) DSS sends query2 to DBO which returns res2 to DSS. DSS can send fail2 or ok to IUI.

4) The four actions close, read, write and replicate can be sent by IUI to DSS, also depending on previous results.

5) If IUI was asking a replicate, then a replicate is performed

6) If IUI was asking a read or write, then rview and wview are sent back from DSS to IUI

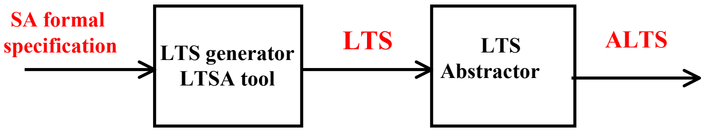
ALTS2



30-25+1=6 independent paths

Tool Support

- LTS can be automatically derived from an Architectural specification (LTS generator Tool, LTSA Tool);
- ALTS can be semi-automatically generated (FC2Tools + The Abstractor Tool)



SA formal specification

LTS generator LTSA tool

LTS

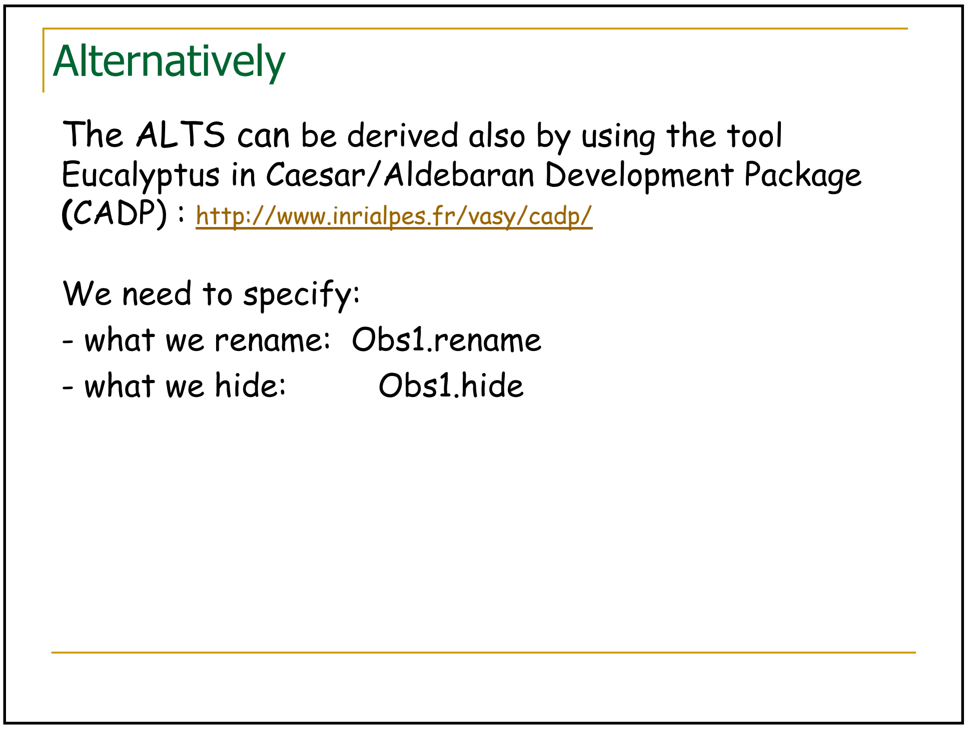
LTS Abstractor

ALTS

FSP : **“Concurrency: State Models & Java Programs” ,** **http://www.doc.ic.ac.uk/~jnm/book/**

LTSA : **http://www-dse.doc.ic.ac.uk/concurrency/**

FC2TOOL : **http://www-sop.inria.fr/meije/verification/**



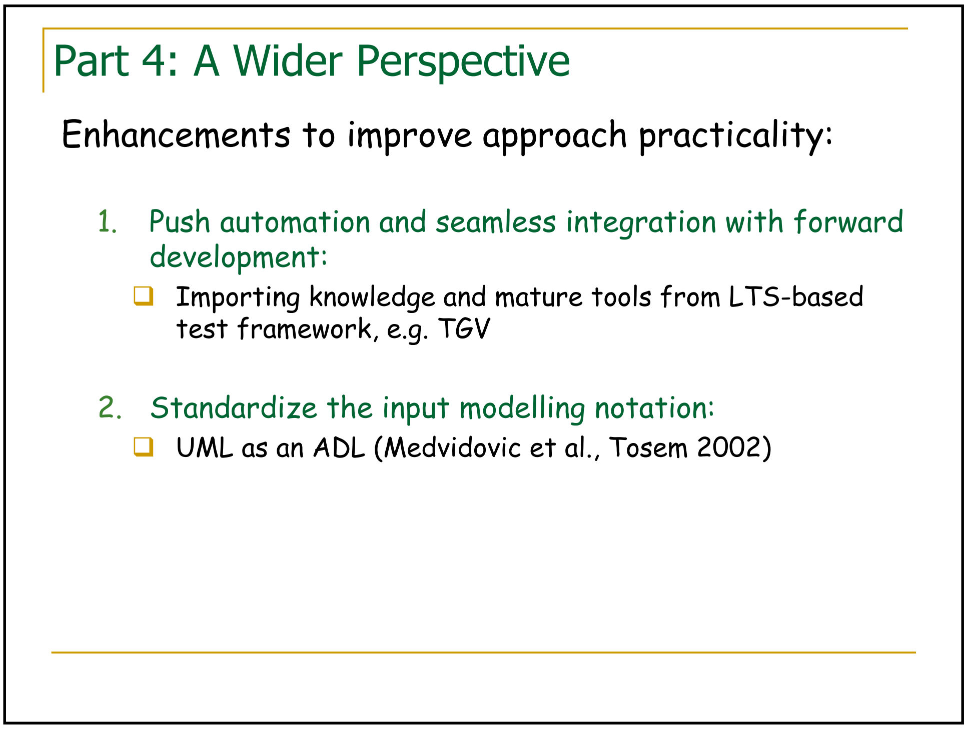
Alternatively

The ALTS can be derived also by using the tool Eucalyptus in Caesar/Aldebaran Development Package (CADP) : http://www.inrialpes.fr/vasy/cadp/

We need to specify:

- what we rename: Obs1.rename

- what we hide: Obs1.hide



Part 4: A Wider Perspective

Enhancements to improve approach practicality:

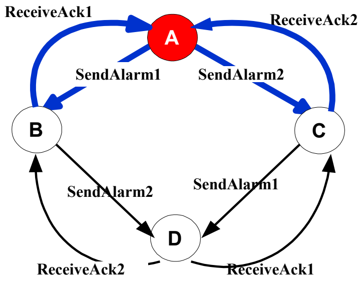
1. Push automation and seamless integration with forward development:
   - ❑ Importing knowledge and mature tools from LTS-based test framework, e.g. TGV

2. Standardize the input modelling notation:
   - ❑ UML as an ADL (Medvidovic et al., Tosem 2002)

On-going work

- Re-using TGV for implementing the ALTS-based technique:
  - Reformulate the "observation" function from the SA view point in terms of TGV notion of "test purpose"
  - Advantage: a complete test graph can be derived, avoiding the round trip LTS -> ALTS -> LTS

TGV-based testing of SA

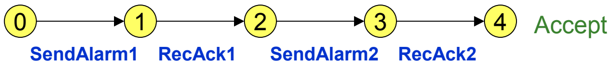


We formerly had:

P3= A B A C A →

SendAlarm1.ReceiveAck1.SendAlarm2.ReceiveAck2

In TGV-based testing, we have to reformulate such an ALTS-path in form of a Test Purpose:



TGV-based testing of SA

- Then, we can:
  - Either generate a Complete Test Graph from the complete LTS see
  - Or directly generate a specified test case, e.g.
- We also need to distinguish between the inputs and the outputs to the system

Test purpose vs. Obs-function:

Roughly, a test purpose would correspond in our approach to a path on the ALTS.

Intuitively, then, we could reformulate our approach within the more mature context of TGV (by translating the *Obs*-function in terms of guidelines for deriving a test purpose)

ALTS offers an intermediate useful step which is missing in TGV

TGV vs. ALTS-based approaches

- Some methodological commonalities
  - Behaviour expressed by means of LTSs
  - Need of algorithms to reduce the many possible LTS observations
- But several conceptual differences
  - Different test levels: SA-based testing tries to capture SA-relevant behaviors, while abstracting away other system functions. Thus, completeness is not the main goal, as in traditional specification-based test approaches
  - Abstraction level: the “distance” between the reference model and the IUT varies much in the two contexts. In SA-based testing, this distance is purposely very high, whereas in specification-based testing, this is often thought as being low.

UML for specifying SAs

- Considering the standard UML, two approaches can be followed: [Medvidovic et al., Tosem 2002]

1. UML “as is”
2. UML Built-in extension mechanisms

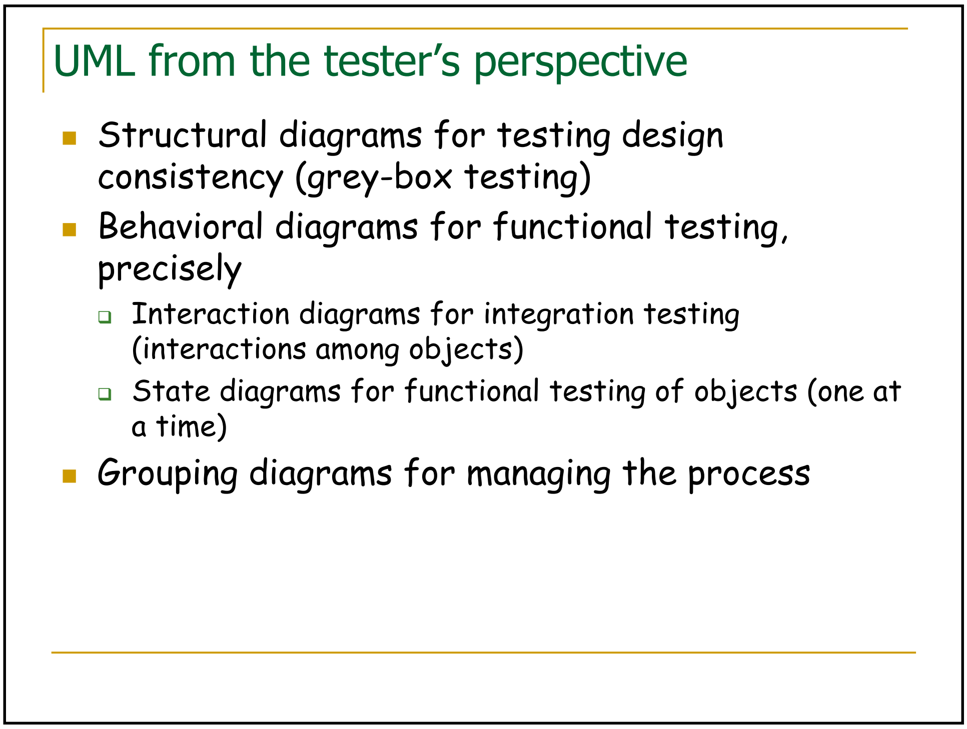
There are several problems with UML-based testing

- UML is a notation, NOT a methodology
- It purposely places few restrictions on usage of the notation
- This flexibility allows for both precision and ambiguity

- However, test from UML is important because of its widespread adoption

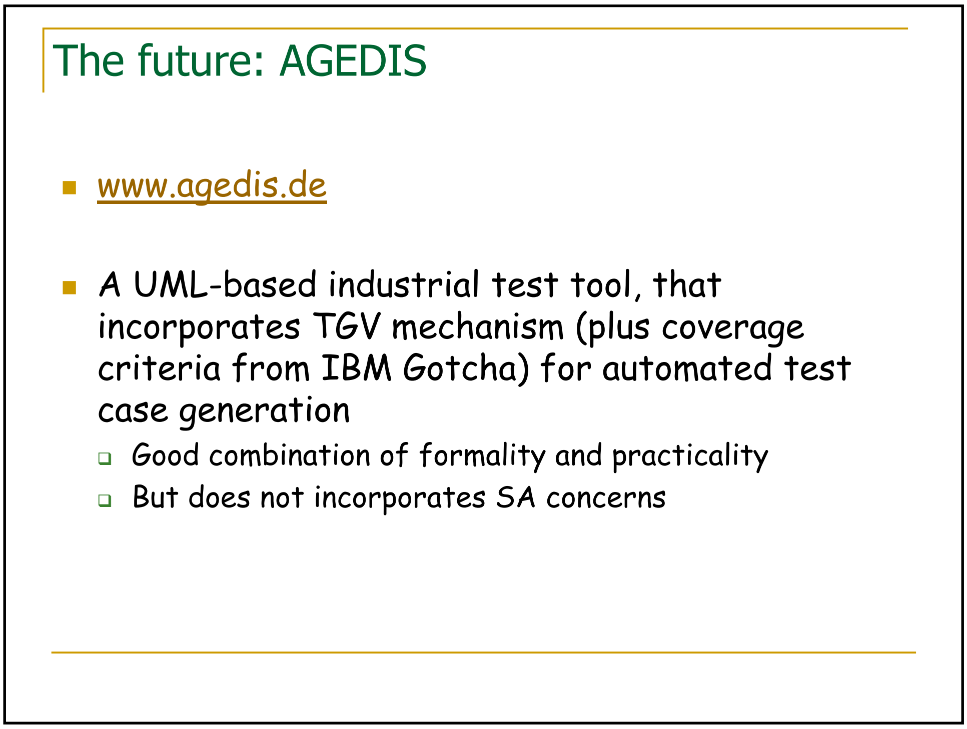
UML & Testing: overview

- still under study: no standardised or generally accepted approach
- the level of formalization and automation achievable depends directly from the level of precision in specification and design
- Binder surveys how each and every UML diagram could be tested (Chap. 8 of his book):
  - Valuable practical insights (but test process too fragmented)
- Several proposals (e.g., Jezequel, Offutt, Briand, Bertolino, ...)– UML test profile - AGEDIS EU project



UML from the tester’s perspective

- Structural diagrams for testing design consistency (grey-box testing)
- Behavioral diagrams for functional testing, precisely
  - Interaction diagrams for integration testing (interactions among objects)
  - State diagrams for functional testing of objects (one at a time)
- Grouping diagrams for managing the process



The future: AGEDIS

- www.agedis.de
- A UML-based industrial test tool, that incorporates TGV mechanism (plus coverage criteria from IBM Gotcha) for automated test case generation
  - Good combination of formality and practicality
  - But does not incorporates SA concerns

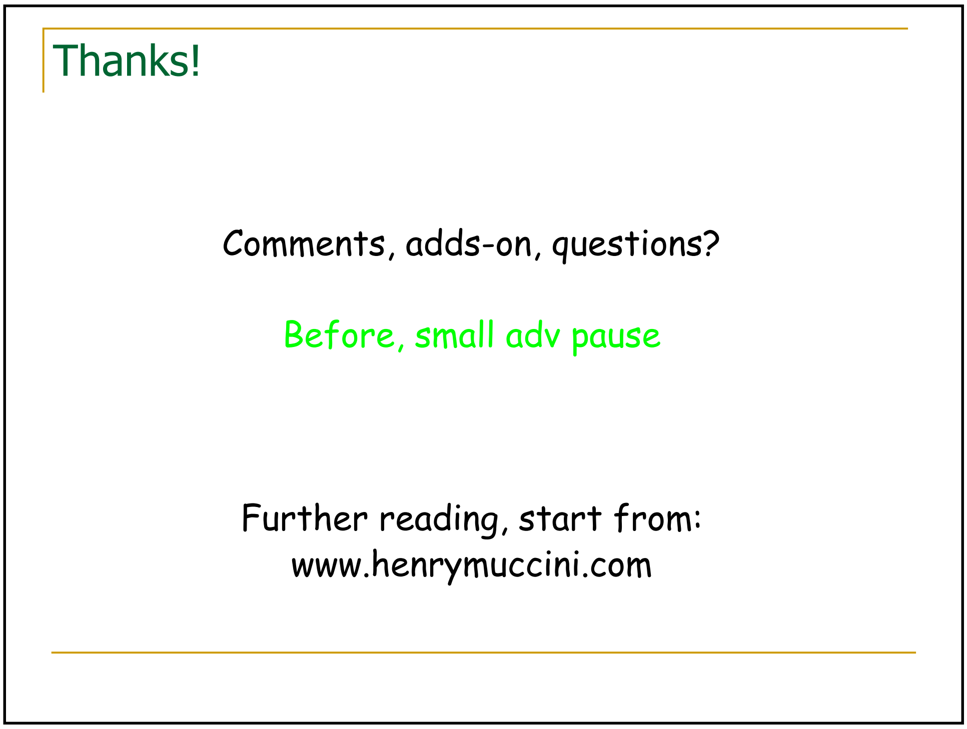
Conclusions

We have surveyed:

- ✓ Basics concepts of software testing, including formal conformance test methods
- ✓ A complete approach proposed for testing a system from an architectural perspective
- ✓ Some examples
- ✓ Proposals for future work

Considerations

- Testing remains an engineering activity
- Nevertheless, formal methods can and should play a role in:
  - Test generation (LTS, ALTS)
  - Validating test results (oracle)
  - Mapping abstract test classes to concrete, executable test cases (through refinement)
- A key issue in SA-based testing is the identification of suitable "test criteria", or ways to look at the system global behavior



Thanks!

Comments, adds-on, questions?

Before, small adv pause

Further reading, start from:
www.henrymuccini.com