UNIVERSITÀ DEGLI STUDI FIRENZE

DINFO Dipartimento di Ingegneria dell'Informazione

Software Engineering for Embedded Systems

Software testing

Laura Carnevali

Software Technologies Lab
Department of Information Engineering
University of Florence
`http://stlab.dinfo.unifi.it/carnevali`
laura.carnevali@unifi.it

Credits

Credits

- These slides are taken from the slides of the lectures of this course given by Prof. Enrico Vicario in the A.Y. 2020/2021
- These slides are authorized for personal use only
- Any other use, redistribution, and profit sale of the slides (in any form) requires the consent of the copyright owners

Outline

Basic concepts

Nature and aims of software testing

- Testing is a method of verification
  - It aims at identifying defects of an Implementation Under Test (IUT) through disciplined experiments and observation of failures
  - It is a dynamic approach (as opposed to static inspection)
- It has inherent limits and capabilities
  - It can prove the presence of defects, but not their absence (E.Dijkstra, "Notes On Structured Programming," 1970)
  - Yet, it is useful in a formative aim
  - It is prescribed in process or product certification of safety-critical systems (e.g., RTCA-D0/178B and RTCA-D0/178C for software in airborne systems) . . . And it is open to integration with formal methods
  - Also, it is a good driver in software development (design for testability)

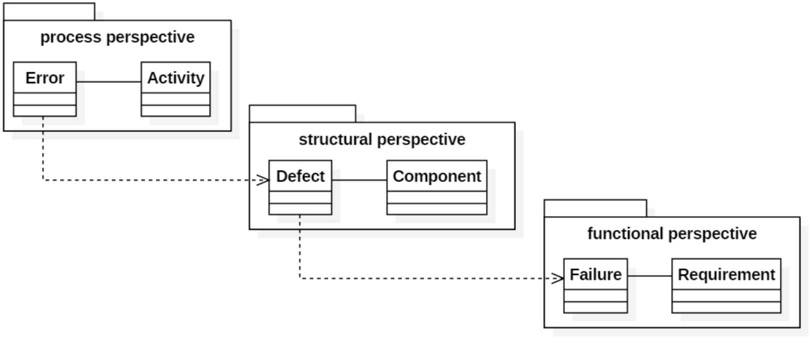
Ontology of defects, failures, and errors

- Three main concepts
  - Failure: a deviation from functional requirements
  - Defect: an element implementation that causes a failure
  - Error: a step in the process that leads to a defect
- An example: the Ariane V flight 501 inquiry board report, 1996
  - Failure: wrong value returned by the SW unit that processes a measurement
  - Defect: insufficient dynamics of the type holding the value may yield overflow
  - Error: the Interface Requirements Specification (IRS) document does not properly identify the actual range of input values

International Software Testing Qualifications Board (ISTQB), "Standard Glossary of Terms used in Software Testing," version 3.2, 2018

Defects and failures, and their ontology (1/3)

- The three concepts belong to different perspectives
  - An **Error** is made in some **Activity** of the development process. . .
  - . . . resulting in a **Defect** in the structure of some **Component**. . .
  - . . . that may lead to a **Failure** w.r.t. some **Functional Requirement**



Defects and failures, and their ontology (2/3)

- Philosophical difficulty of defining a fault
    - Conceptual definition of a fix is easier than that of the fault
- Specific nature of Sw testing
    - Much less related to aging and physical production
    - Most faults are there since the design and coding

Defects and failures, and their ontology (3/3)

- A different ontology is applied in Dependability Engineering
  - A **Fault** leads a subsystem in a state of **Error**...
  - ...which may lead to the subsystem **Failure**...
  - ...which in turns becomes a **Fault** for the higher-level **System**
- e.g. Ariane V flight 501
  - Fault: wrong value returned by the measurement processing unit
  - Error: inconsistent state in the angular acceleration control unit
  - Failure: shutdown of the prime processor propagated
    to the higher-level system as a Fault...
- Main differences between the two perspectives
  - Emphasis on state behavior and system/subsystem hierarchy
  - Errors may be recovered, opening the way to Fault tolerance

A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, 2004

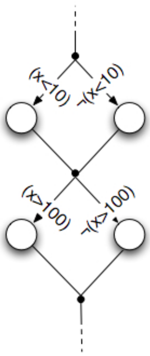
Activities in the methodology of testing

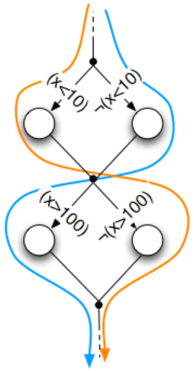
- Test case selection
  - Defines a suite of tests able to reveal (cover) possible Faults
- Input generation
  - Identifies inputs that let the system run along a test case
- Test case execution
  - Drives the execution of the test suite
    (drivers and stubs, scripts, scaffolds, logs, . . . Junit, Mockito)
- Oracle verdict
  - Decides whether the IUT passes a test,
    i.e., whether the test results comply with functional requirements
- Debugging
  - Traces back observed functional failures to structural faults
- Coverage analysis
  - Evaluates how much the test results cover the space of behaviors of the IUT

Remarks on test case selection

- It identifies a suite of tests able to reveal (cover) possible Faults
- It is the most characterizing step of the testing methodology
- It relies on some abstraction of the IUT... and on some coverage criterion

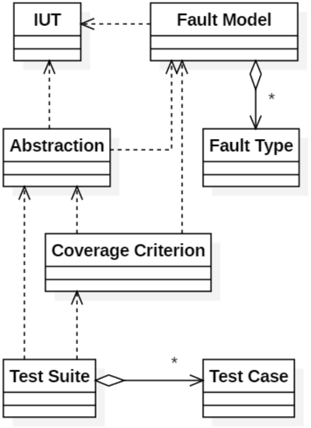
```
if(x<10){...}
else{...}
if(x>100){...}
else{...}
```





Fault model

- **Abstraction** and **Coverage Criterion** depend on some **Fault Model**
  - It is a set of **Fault Types** in the **IUT**,
    i.e., the typed of faults that are being fought
  - It in turn depends on the IUT structural and functional characteristics



Functional perspective vs structural perspective (1/2)

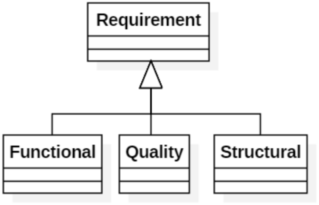
- Abstractions can take different perspectives

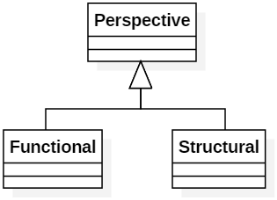
- **Functional** (black box) refers to the IUT specification
  - E.g., use case diagrams, System Requirements Specification (SRS) prescribed by the MIL-STD-498 standard, conceptual class diagram, ...
- **Structural** (white box) refers to the IUT implementation
  - E.g., class diagrams, System Design Description (SDD) prescribed by the MIL-STD-498 standard, source code or binary code, ...
- **Architectural** (grey box) refers to the IUT architecture
  - E.g., architectural design, System Subsystem Design Description (SSDD) and Interface Requirements Specification (IRS) of the MIL-STD-498 standard, ...

- Model Driven Development (MDD) may obfuscate the concept, e.g., automated code generation from a requirements model

Functional perspective vs structural perspective (2/2)

- About the meaning of “functional” in Software Engineering
  - Functional, structural, and quality requirements . . .
  - . . . with quality characteristics defined by the ISO/IEC 9126 standard, now replaced by the ISO/IEC 25010:2011 standard. . .
  - . . . and classified as internal, external, or in-use
- In Software Testing:
  - The functional perspective refers to any requirement
  - The structural perspective refers to any aspect of the implementation

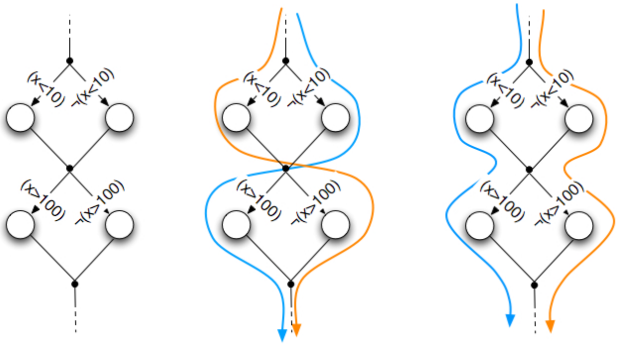




Remarks on input generation

- It identifies inputs that let the system run along a test case
- It is a kind of undecidable problem
- It may be not feasible
  - Due to unfeasible paths (false behaviors) added in the abstraction
- It is coupled with test case selection
  - A Coverage Criterion can be implemented by different Test Suites, with different conditions of path feasibility

```
if(x<10){...}
else{...}
if(x>100){...}
else{...}
```



Remarks on oracle verdict

- It decides whether the IUT passes a test
  i.e., whether the test results comply with the functional requirements
- The oracle verdict may be inconclusive
  - Due to the fact that the IUT response may be not fully controllable
- It is the most hard part to automate
  - It requires an executable representation of requirements
    e.g., use a MatLab module as specification for a c program
  - It is often performed manually in a tautological manner:
    judge the test results in the absence of a specification

Remarks on debugging

- It traces back the observed functional failures to structural faults
- A Failure may be associated with multiple Faults
  - A malfunction may result from the interaction of multiple details of the IUT. . .
  - . . . and it can be remove through joint application of multiple fixes. . .
  - . . . which then raises the issue of regression testing

Remarks on coverage analysis

- It evaluates how much the test results cover the space of behaviors of the IUT
- Realization similar to that of test case selection
  - Extent to which test results cover some IUT abstraction
  - If the IUT passes a test suite $\Rightarrow$ A kind of justifiable degree of confidence on the absence of residual defects in the IUT
  - Based on the degree of coverage of some Abstraction
  - Subtends assumptions on the size of the Abstraction
  - Relies on some code instrumentation (e.g., aspect oriented)
- Test case selection & coverage analysis can mix perspectives
  - An example: select test cases from use cases, and then evaluate coverage on an abstraction based on the code (e.g., RTCA-DO/178B, RTCA-DO/178C)

Control flow testing

Ccntrol flow testing: introduction

- Provides an abstraction, i.e., the Control Flow Graph (CFG)
- Provides a suite of criteria, i.e., all nodes, all edges,. . . all paths, boundary interior,. . . Modified Condition Decision Coverage (MCDC)
- Applicable both for test case selection and for coverage analysis
- Applicable both in the structural perspective and in the functional perspective

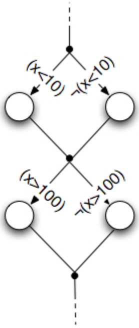
Control Flow Graph (CFG) (1/2)

- Vertices can be **Lines of Code (LOCs)** or **basic blocks** (Rapps Weyucker)
  - A basic block is a maximal set of statements $\langle S_1, \dots S_N \rangle$ such that:
    - S_N is the unique executional successor of S_{N-1},
    - S_{N-1} is the unique executional predecessor of S_N

  i.e., a max set of statements that are executed always together

  i.e., no label target, no branches, break, return, ... maximal
- Edges are the relation of executional succession

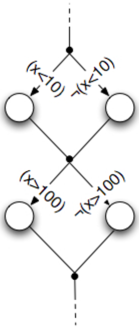
```
if(x<10){...}
else{...}
if(x>100){...}
else{...}
```



Control Flow Graph (CFG) (2/2)

- The CFG abstraction removes data dependencies

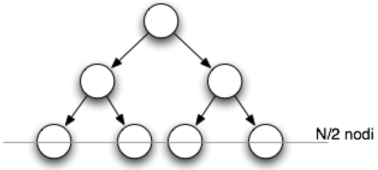
```
if(x<10){...}
else{...}
if(x>100){...}
else{...}
```



- Subtended **fault model**
  - Represents defects affecting control guards or control variables used in guards, that will let the IUT deviate from the expected control flow
- Applicable in different abstraction grains
  - LOCs, basic blocks, function calls, services invokations, ...

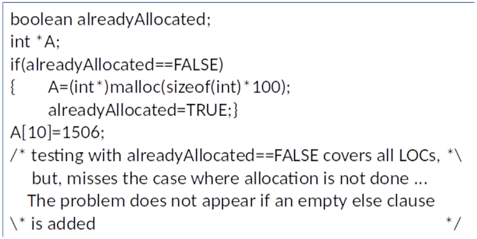
Coverage criteria: all-nodes

- All-nodes (a.k.a. block coverage, statement coverage)
  - Visit each vertex of the CFG
  - Minimum coverage required to ensure that each Line of Code (LOC) has been executed at least once during testing
  - Yet, sufficient for certification of a large class of safety-critical systems
- Complexity $O(N)$ where N is the number of nodes
  - In terms of the number of test cases
  - ...devise a program where ...



Coverage criteria: limits of all-nodes

- Does not guarantee coverage of all edges

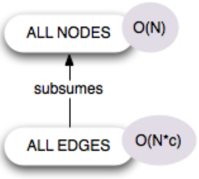


```
boolean alreadyAllocated;
int *A;
if(alreadyAllocated==FALSE)
{      A=(int*)malloc(sizeof(int)*100);
       alreadyAllocated=TRUE;}
A[10]=1506;
/* testing with alreadyAllocated==FALSE covers all LOCs, *\
    but, misses the case where allocation is not done ...
   The problem does not appear if an empty else clause
\* is added                                              */
```

- Does not distinguish how a cycle was left (guard, break, return, goto)
- Does not depend on the number of iterations in a cycle, and completely ignores a do-while structure
- In coverage analysis, may be not proportional to complexity (unbalanced number of LOCs on different branches)

Coverage criteria: all-edges

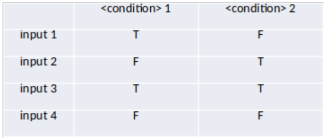
- All-edges (a.k.a. all-decisions)
  - Traverse each edge of the CFG
- All-edges subsumes all-nodes
  - May differ with joint presence of branch and confluence
- Complexity is still $O(cN)$ where c is the max output degree of statements
  - $O(N)$ reaches each node, and from each node, c cases cover all outputs
- Limits of all-edges
  - Does not guarantees that guards are tested under all combinations that lead to the same decision
  - May be relevant if guards produce side effects



Coverage criteria: multiple conditions (1/2)

- Extends all-edges by covering each decision under all different conditions of the guard
    - A condition is a maximal expression without Boolean connectives
- Becomes relevant when condition guards may produce side effects

```
if(x>10&&y<3)
```



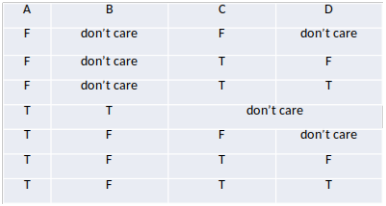
	<condition> 1	<condition> 2
input 1	T	F
input 2	F	T
input 3	T	T
input 4	F	F

- . . . in conjunction with short-circuits on expressions

Coverage criteria: multiple conditions (2/2)

- But, complexity is $O(cN2^k)$ where k is the number of conditions in a guard
- For languages with short-circuited expressions, many cases are equivalent

```
if((A && B) || (C && D))
```



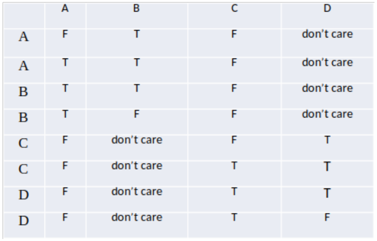
A	B	C	D
F	don't care	F	don't care
F	don't care	T	F
F	don't care	T	T
T	T	don't care	
T	F	F	don't care
T	F	T	F
T	F	T	T

- this holds for c, c++, Java, but not for Ada

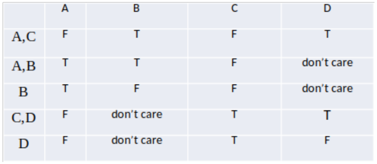
Coverage criteria: modified condition decision coverage (1/2)

- Modified Condition Decision Coverage (MCDC)
  - Extends all-edges by requiring that each decision be covered so that in some test each condition determines each different decision in both ways

```
if((A && B) || (C && D))
```



	A	B	C	D
A	F	T	F	don't care
A	T	T	F	don't care
B	T	T	F	don't care
B	T	F	F	don't care
C	F	don't care	F	T
C	F	don't care	T	T
D	F	don't care	T	T
D	F	don't care	T	F



	A	B	C	D
A,C	F	T	F	T
A,B	T	T	F	don't care
B	T	F	F	don't care
C,D	F	don't care	T	T
D	F	don't care	T	F

Coverage criteria: modified condition decision coverage (2/2)

- For short-circuited languages, MCDC is equivalent to multiple-conditions
- Complexity is $O(ckN)$
  - c cases for each condition
  - Analysis for identification of test cases is $O(2^k)$
- Prescribed since RTCA/DO 178B
  - Developed by Boeing and assumed by RTCA (Radio Technical Commission for Aeronautics, US) and the European Org for Civil Aviation Equipment

Radio Technical Commission for Aeronautics, DO-178B, “Software Considerations in Airborne Systems and Equipment Certification,” 1992.

Coverage criteria: all paths

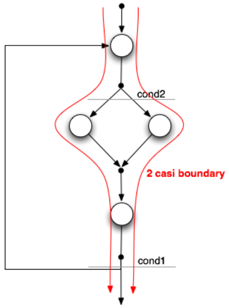
- All paths (a.k.a. all predicates)
  - Each different path is covered by at least one test
  - Complexity is $O(2^N)$ if no cycle exists, unbounded with cycles
  - Is a concept, not a practice
  - May become affordable on very coarse abstractions
- Still, might be non revealing, due to incidental correctness

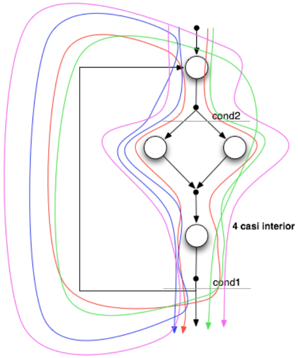
Coverage criteria: boundary interior and structured path

- Boundary interior selects a finite subset of all paths
  by covering classes of equivalence in the set of paths
  - Boundary tests are paths that traverse the cycle only once
    with different tests for each path in the cycle
  - Interior tests are paths that traverse the cycle multiple times
    with different tests for each path in the first cycle iteration
- Structured path testing extends the concept from 1 to k iterations

Coverage criteria: boundary interior - an example

```
do
{ ...
  if (condition2)
  {     statement1
  } else
  {     statement2
  }
  ...
} while (condition1);
```

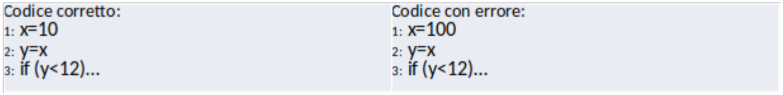




Data flow testing

Data flow testing

- How faults can propagate to observable failures
  - A faulty side effect on some variable is not observed until the variable is used
  - The fault can activate a failure or be propagated to some other variable



```
Codice corretto:          Codice con errore:
1: x=10                   1: x=100
2: y=x                    2: y=x
3: if (y<12)...           3: if (y<12)...
```

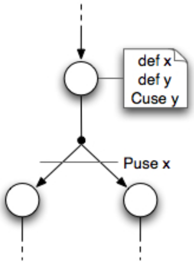
- Exercise some paths from where a variable is side-effected up to where it is used, i.e., cover data coupling among statements
- Applicable both for test case selection and coverage analysis
- First in the structural and then in the functional perspective

Data Flow Graph (DFG)

- Extends the CFG with **def-x** and **use-x** annotations
  for each **relevant** variable x (not for all variables)
- Cover the DFG with a suite of criteria: all-defs, all-uses, all-DU-paths
- Both for test case selection and coverage analysis
- First in the structural and then in the functional perspective

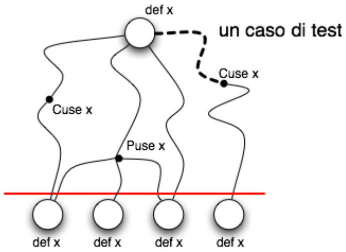
Data Flow Graph (DFG): annotations and concepts

- A def-x is a location where variable x is side effected
- A use-x is a location where the value of x is used
  - A c-use-x is a use occurring out of any guard (computational)
  - A p-use is a use occurring within a guard (predicate)
- Distinction of p-use and c-use
  - Does not reflect the syntax, is more about consequences: a p-use can let the flow diverge, a c-use can propagate
- A def-clear-path-x is an acyclic path that initiates with a def-x, terminates with a use-x, and does not traverse any def-x



Coverage criteria: all-def

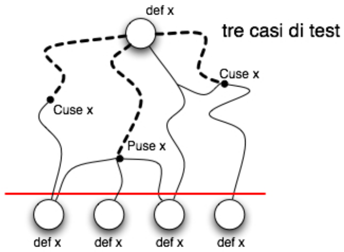
- All definition coverage: for each def, at least one path until some use



- Program slicing: repeat this for each **relevant** variable x

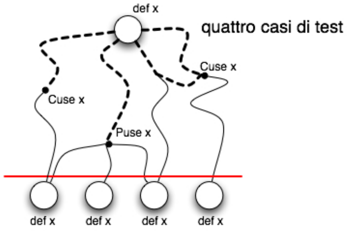
Coverage criteria: all-uses

- All uses coverage:for each def, at least one path for each use reachable through some def-clear-path



Coverage criteria: all-DU-paths

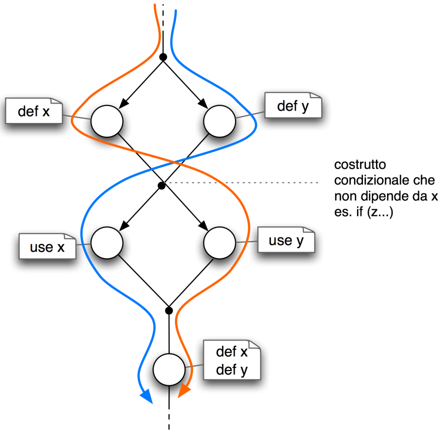
- All def use paths: all def-clear-paths from each definition to each use reached by that definition and each successor node of the use



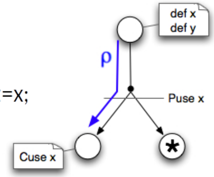
- Remark: DU paths are acyclic by definition

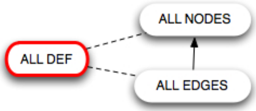
Coverage criteria: control flow criteria vs data flow (1/4)

- All-def is not comparable with all-nodes or all-edges
  - all def does not include all nodes
  - but, it is not included by all edges



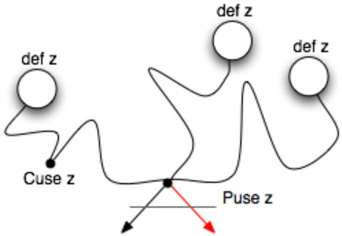
```
x=10;
y=20;
if (y>100) z=x;
else z=y;
```

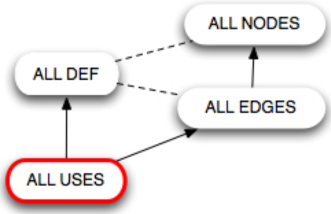




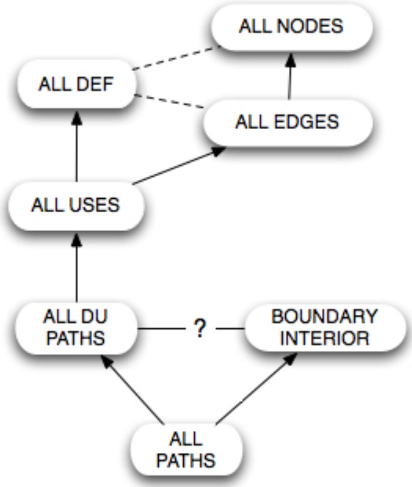
Coverage criteria: control flow criteria vs data flow (2/4)

- All-uses subsumes all-edges
  - From any c-use, move backward and find a def;
    there will be at least a path for the pair def-use
    (not necessarily the path you found moving backward);
    if no def is found, a compiler warning can be given
  - From any p-use, move backward starting from the decision node



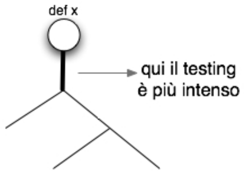


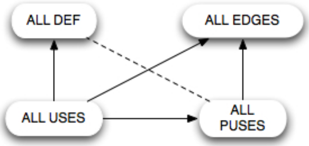
Coverage criteria: control flow criteria vs data flow (3/4)

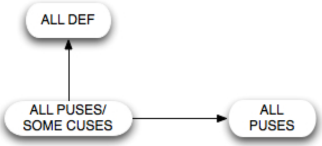


Coverage criteria: control flow criteria vs data flow (4/4)

- All-p-uses restricts all uses to cover pairs def to p-use
- All-p-uses-some-c-uses extends all p-uses to cover all-def
  - Reduces over-coverage of defs while maintaining comparability

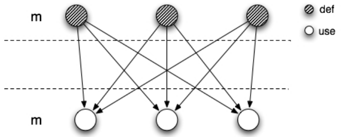


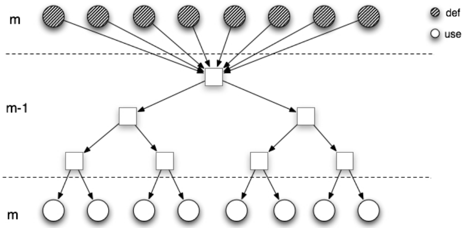




Coverage criteria: complexity

- All-uses is $O(hN^2)$ where h is the maximum out degree of a statement
- Sufficient (left figure): pairs are no more than $O(N^2)$
- Necessary (right figure)
    - $O(N^2)$ pairs def-use in a graph
    - There is not any program with this CFG
    - But, we can define a program where all uses requires $O(N^2)$





On structural testing vs functional testing

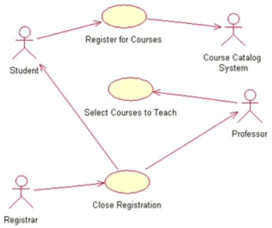
- Structural testing is apparently more grounded
  - Implementation is more formally specified than requirements
- But, structural test case selection is tautological
  - Example: `x=100 ... if(x>10)` → `x=1 ... if(x>10)` → fault covered
    `x=100 ... if(x>10)` → `y=100 ... if(x>10)` → fault not covered
  - Cannot detect missing functions
- Implementation is more complex than requirements
  - Less accessible to people not involved in development
- A virtuous combination
  - Functional test case selection
  - Structural coverage analysis

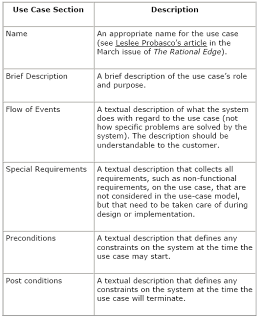
Control and data flow testing in functional perspective

- Control and data flow testing can be applied also in functional perspective
  - Whenever the specification subtends some kind of control flow, with possible explicit representation of def/use dependencies
- Some notable examples
  - The page navigation diagram of a web application
  - A data flow diagram specification
  - The flow of events in a Use Case Template
  - The concurrent flow of an Activity Diagram
  - Any executable specification (e.g., a Matlab prototype)

Generating test cases from use cases (1/3)

- Use case diagrams
  - Actors and use cases; stereotypes `<<includes>>`, `<<extends>>`, ...
  - Abstraction level: user goal
- Use case templates
  - A textual specification of use cases, more or less "dressed"
  - Always includes a flow of events (user/system dialog, normal/alternative flows)
- Not only for object oriented development
  - Fits well in a Software Requirements Specification (SRS - MIL-STD-498)

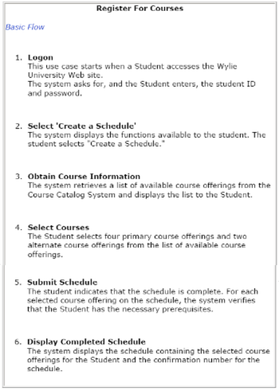




Use Case Section	Description
Name	An appropriate name for the use case (see Leslee Probasco's article in the March issue of *The Rational Edge*).
Brief Description	A brief description of the use case's role and purpose.
Flow of Events	A textual description of what the system does with regard to the use case (not how specific problems are solved by the system). The description should be understandable to the customer.
Special Requirements	A textual description that collects all requirements, such as non-functional requirements, on the use case, that are not considered in the use-case model, but that need to be taken care of during design or implementation.
Preconditions	A textual description that defines any constraints on the system at the time the use case may start.
Post conditions	A textual description that defines any constraints on the system at the time the use case will terminate.

Generating test cases from use cases (2/3)

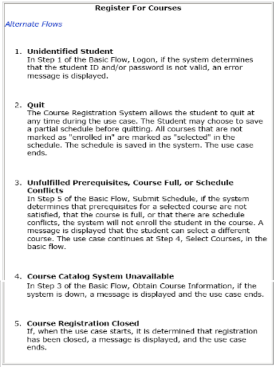
- Basic and alternative flows identify a Control Flow Graph (CFG)
  - Basic and alternative flows also identify a Data Flow Graph (DFG), when data are defined/used at different steps of the flow



Register For Courses

Basic Flow

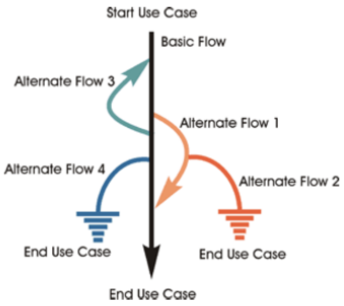
1. **Logon**
   This use case starts when a Student accesses the Wylie University Web site.
   The system asks for, and the Student enters, the student ID and password.
2. **Select 'Create a Schedule'**
   The system displays the functions available to the student. The student selects "Create a Schedule."
3. **Obtain Course Information**
   The system retrieves a list of available course offerings from the Course Catalog System and displays the list to the Student.
4. **Select Courses**
   The Student selects four primary course offerings and two alternate course offerings from the list of available course offerings.
5. **Submit Schedule**
   The student indicates that the schedule is complete. For each selected course offering on the schedule, the system verifies that the Student has the necessary prerequisites.
6. **Display Completed Schedule**
   The system displays the schedule containing the selected course offerings for the Student and the confirmation number for the schedule.



Register For Courses

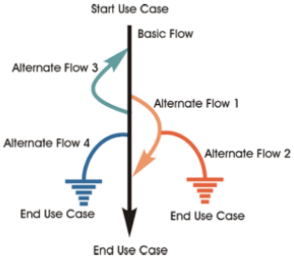
Alternate Flows

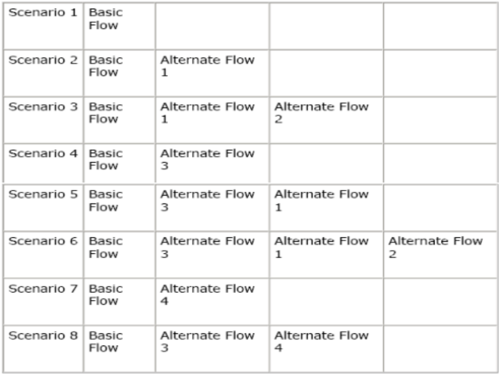
1. **Unidentified Student**
   In Step 1 of the Basic Flow, Logon, if the system determines that the student ID and/or password is not valid, an error message is displayed.
2. **Quit**
   The Course Registration System allows the student to quit at any time during the use case. The Student may choose to save a partial schedule before quitting. All courses that are not marked as "enrolled in" are marked as "selected" in the schedule. The schedule is saved in the system. The use case ends.
3. **Unfulfilled Prerequisites, Course Full, or Schedule Conflicts**
   In Step 5 of the Basic Flow, Submit Schedule, if the system determines that prerequisites for a selected course are not satisfied, that the course is full, or that there are schedule conflicts, the system will not enroll the student in the course. A message is displayed that the student can select a different course. The use case continues at Step 4, Select Courses, in the basic flow.
4. **Course Catalog System Unavailable**
   In Step 3 of the Basic Flow, Obtain Course Information, if the system is down, a message is displayed and the use case ends.
5. **Course Registration Closed**
   If, when the use case starts, it is determined that registration has been closed, a message is displayed, and the use case ends.



Generating test cases from use cases (3/3)

- Tests can be selected by coverage of the CFG or DFG
  - Coarser than structural abstractions and usually acyclic
  - Coverage can afford "expensive" criteria
  - Also functional criteria (e.g., for non-deterministic cycles)
- A test case can cover multiple flows
- A user-level scenario can cover multiple flows of multiple use cases

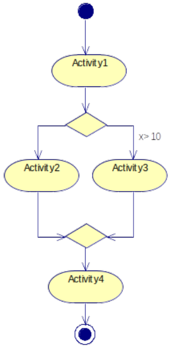


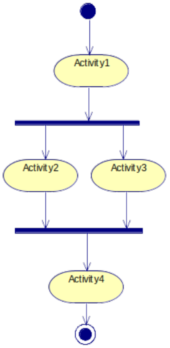


Scenario 1	Basic Flow			
Scenario 2	Basic Flow	Alternate Flow 1		
Scenario 3	Basic Flow	Alternate Flow 1	Alternate Flow 2	
Scenario 4	Basic Flow	Alternate Flow 3		
Scenario 5	Basic Flow	Alternate Flow 3	Alternate Flow 1	
Scenario 6	Basic Flow	Alternate Flow 3	Alternate Flow 1	Alternate Flow 2
Scenario 7	Basic Flow	Alternate Flow 4		
Scenario 8	Basic Flow	Alternate Flow 3	Alternate Flow 4	

Generating test cases from activity diagrams (1/4)

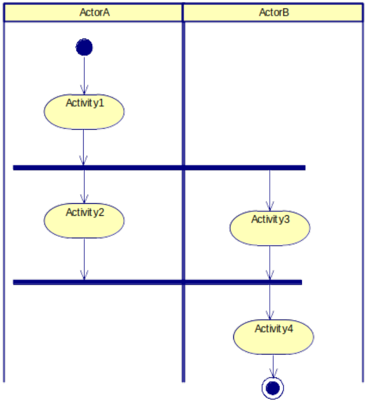
- UML Activity Diagrams (ADs) capture **control flow** and **concurrency**
  - Activity, sequence, start/end, branch, fork/join
- Derived from the conceptual models of Petri nets
  - Places and tokens, transitions, pre-conditions and post conditions, firing
- Derived from the Specification and Description Language (SDL)
- Well suited for specifying procedures, processes, workflows, business modeling
- Key diagrams in the Systems Modeling Language (SysML)

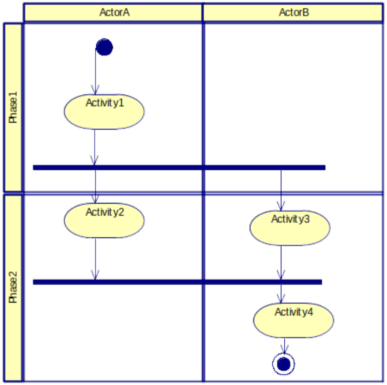




Generating test cases from activity diagrams (2/4)

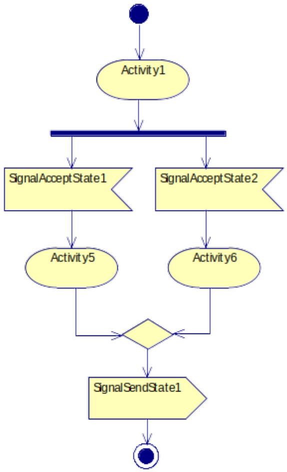
- **Swimlanes** allocate **Activities** to **Actors** and **Resources**
  - Multiple partitions (2 in the example)





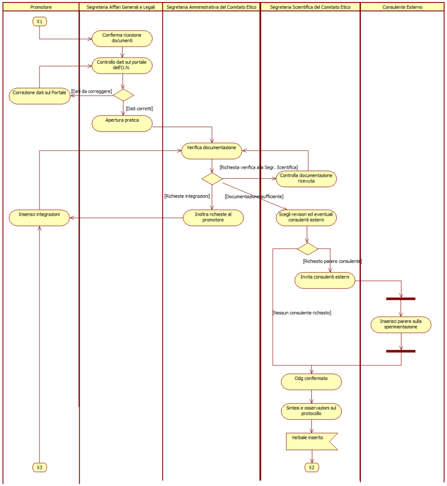
Generating test cases from activity diagrams (3/4)

- Messages permits encoding of signals exchanged by multiple processes (as in the semantics of the SDL)



Generating test cases from activity diagrams (4/4)

- An AD is naturally abstracted as a DFG
  - Specifying the control flow
  - Specifying def/use dependencies among subsequent or concurrent activities


Promotore
Segreteria Affari Generali e Legali
Segreteria Amministrativa del Comitato Etico
Segreteria Scientifica del Comitato Etico
Consulente Esterno
X1
Conferma ricezione documenti
Controllo dati sul portale dell'O.N.
Correzione dati sul Portale
[Dati da correggere]
[Dati corretti]
Apertura pratica
Verifica documentazione
[Richiesta verifica alla Segr. Scentifica]
Controlla documentazione ricevuta
[Richieste integrazioni]
[Documentazione sufficiente]
Inserisci integrazioni
Inoltra richieste al promotore
Scegli revisori ed eventuali consulenti esterni
[Richiesto parere consulente]
Invita consulenti esterni
[Nessun consulente richiesto]
Inserisci parere sulla sperimentazione
Odg confermato
Sintesi e osservazioni sul protocollo
Verbale inserito
X3
X2

Unit vs integration testing

- Unit testing is about single components
  - Often taking a structural perspective,
    being dependent on design decomposition
  - Think of using Junit to test single java classes,
    or using Cantata to test single functions in a c file
  - Performed during development, by the same people who developed and understand the code, unless forbidden by process certification requirements
  - With no contractual value, unless prescribed by the process model
  - Often with formative aim, also in test first programming approach
- Integration testing is about multiple components
  - Often taking a functional perspective
  - Often with contractual aim, as in acceptance testing

Finite state testing

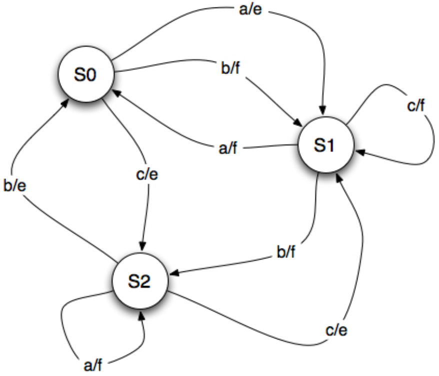
Reactive systems

- Maintain an ongoing interaction with the environment, often only partially predictable, while ensuring functional requirements
  - Contrasted with transformational systems
- An abstraction for a large class of **cyber-physical systems**
  - Examples: an operating system, a control system, an embedded software component, . . . a user interface
  - Demand for specification formalisms and verification methods

Z. Manna, A. Pnueli, "The temporal logic of reactive and concurrent systems: Specification.", Springer Science & Business Media, 2012.

Finite state machines

- In the industrial practice, a specification often revolves around a finite state machine defined as $\langle S, I, O, E \rangle$
  - A set of **states** $S := \{S_0, S_1, S_2\}$ with an initial state $S_0 \in S$
  - A set of **inputs** $I := \{a, b, c\}$
  - A set of **outputs** $O := \{e, f\}$
  - A set of **transitions** $E \subseteq S \times I \times O \times S$



Implementing and testing finite state machines

- Consequences on the structure of the IUT...
  - State-dependent functions attached to transitions...
  - Locations and goto, ...state pattern
- ...and on testing objectives and methodology
  - **Conformance testing**: check whether the IUT behaves according to a specified Finite State Machine (FSM)

Conformance testing for a finite state machine

- Equivalence between a specification S and an IUT I
  - States s in S and i in I are **V-equivalent** if, starting from s and i, the same input sequence of length V produces the same output sequence
  - States s and i are equivalent, if they are V-equivalent for any V
  - FSMs S and I are equivalent if their initial states are equivalent
- Fault Model
  - Output fault: the next state is correct, but the output is not
  - Transfer fault: the output is correct, but the next state is not
  - Output and transfer faults can happen jointly
  - Orthogonal to the Fault Model of control/data flow testing

Conformance testing through the WP-method

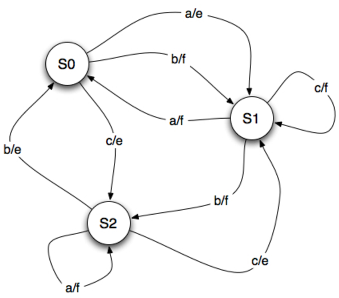
- The specification is an FSM
  - Completely specified: accepts all inputs in each state
  - Observable: in any state, some output is emitted for each input
  - Deterministic: 1 transition and output for each state and input
- The IUT behaves as some not-identified FSM
  - Deterministic, completely specified, and observable
  - Reset is correct, i.e. it always leads to the same state
  - The same number of states as in the specification
  - . . . but, the state is not observable, only the outputs are
- A kind of black-box testing

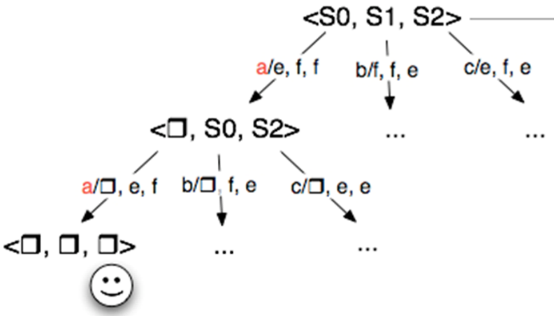
The W-method: S. Fujiwara, G. VonBochmann, F. Khendek, M. Amalou, A. Ghedamsi, "Test selection based on finite state models," IEEE Transactions on software engineering, 1991.

A minor extension of the W-method: T.S.Chow, "Testing software design modeled by finite-state machines," IEEE transactions on software engineering, 1978

Identification (1/2)

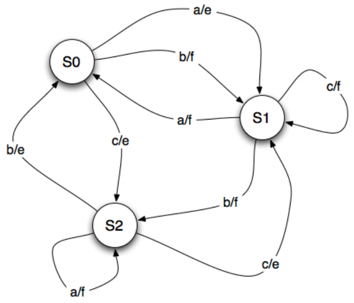
- The problem of identification
  - After reset, the IUT is in some un-identified state in $\{S_0, S_1, S_2\}$
  - Application of different inputs can disambiguate initial state and visited states, depending on the observed outputs

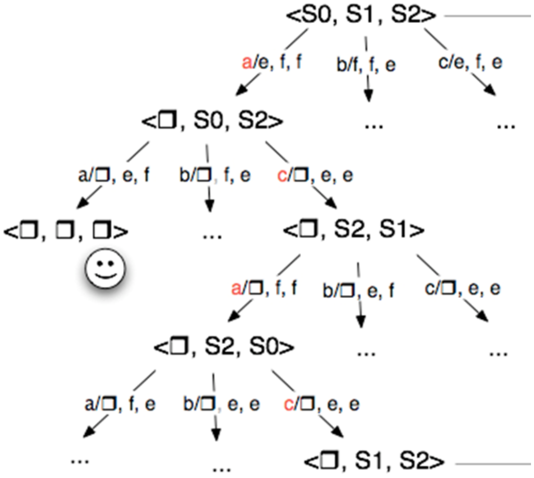




Identification (2/2)

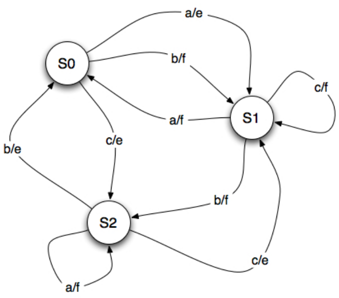
- Is it possible that the sequence never comes to the end?
  - A strict subset will eventually be encountered, detecting the unbounded cycle
  - Termination proof is much similar to the lemma by Karp & Miller for Petri nets

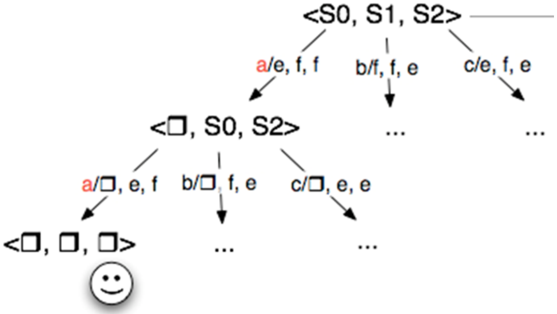




The W-method: characterization set (1/2)

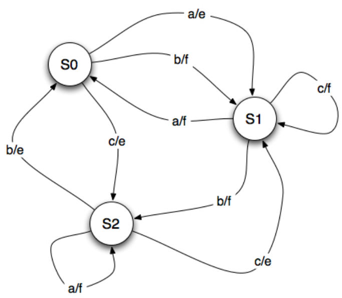
- **Step 1**: select a **characterization set** W
  - A set of input sequences that disambiguate the initial state
  - E.g., $W = \{\{a\}, \{b\}\}$ represents 2 sequences of length 1
  - E.g., $W = \{\{aa\}\}$ represents 1 sequence of length 2
  - E.g., $W = \{\{ab\}\}$ represents another sequence of length 2

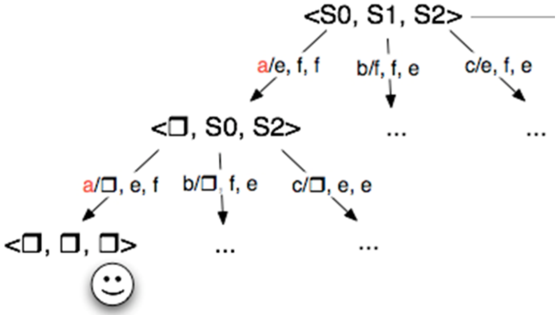




The W-method: characterization set (2/2)

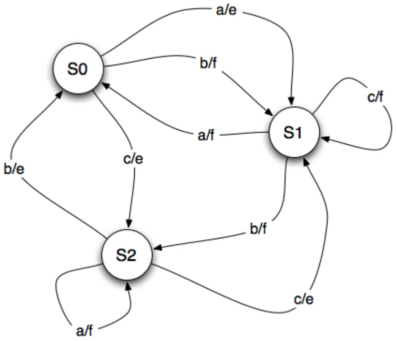
- Trade-off between length and number of sequences
  - E.g. $W = \{\{a\}, \{b\}\}$ versus $W = \{\{aa\}\}$
  - Much depending on the complexity of reset (which can be expensive for a cyber-physical system)
  - May become relevant with extra-states (see later)
  - A single sequence may be not always feasible





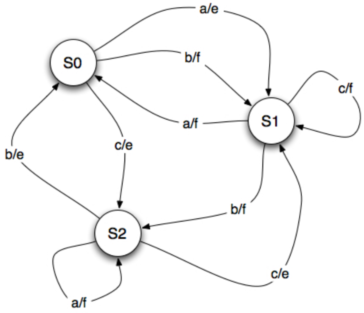
The W-method: state cover (1/4)

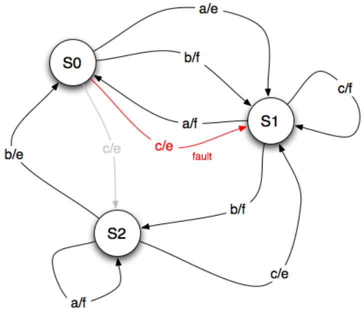
- **Step 2**: select a test suite Q with a test case in each state of the specification (starting after a reset)
  - E.g., $Q = \{\{-\},\{b\},\{c\}\}$
- Obtain a state cover with the suite $Q \times W$
  - All concatenations of any element of Q and any element of W
  - E.g., for $W = \{\{a\},\{b\}\}$, $Q \times W = \{\{-\},\{b\},\{c\}\} \times \{\{a\},\{b\}\} = \{\{a\},\{b\},\{ba\},\{bb\},\{ca\},\{cb\}\}$



The W-method: state cover (2/4)

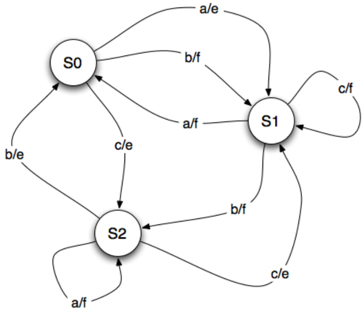
- If the IUT passes the test suite $Q \times W$, then each state of the IUT is uniquely and correctly identified
  - Q is built so as to terminate in each state of the specification
  - suffix W guarantees that state correspond to that in the IUT
  - E.g., $Q \times W = \{\{-\}, \{b\}, \{c\}\} \times \{\{a\}, \{b\}\}$
    $= \{\{a\}, \{b\}, \{ba\}, \{bb\}, \{ca\}, \{cb\}\}$
  - E.g., transfer fault in S_0 revealed by $\{\{c\}\} \times W$, as W tells S_1 from S_2

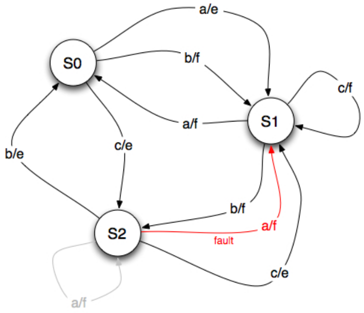




The W-method: state cover (3/4)

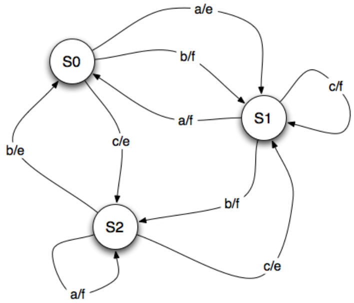
- If the IUT passes the test suite $Q \times W$, then each state of the IUT is uniquely and correctly identified
  - Q is built so as to terminate in each state of the specification
  - suffix W guarantees that state correspond to that in the IUT
  - E.g., $Q \times W = \{\{-\}, \{b\}, \{c\}\} \times \{\{a\}, \{b\}\}$
    $= \{\{a\}, \{b\}, \{ba\}, \{bb\}, \{ca\}, \{cb\}\}$
  - E.g., transfer fault in S_2 revealed by $\{\{c\}\} \times W$, as W tells S_1 from S_2





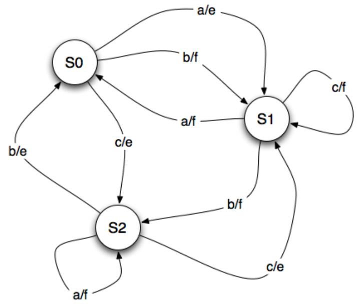
The W-method: state cover (4/4)

- **Step 3**: select a test suite P with a test case traversing each edge and terminating after that
  - E.g., $Q = \{\{a\}, \{b\}\}$
  - E.g., $P = \{\{-\}, \{a\}, \{b\}, \{c\}, \{ba\}, \{bb\}, \{bc\}, \{ca\}, \{cb\}, \{cc\}\}$
- Obtain a transition cover with the suite $P \times W$
  - All concatenations of any element of P and any element of W
- The transition cover will reveal any transfer or output fault in each edge of the specification



The W-method: fault coverage

- The W-Method guarantees **full coverage** of output or transfer faults in the IUT, . . . provided that the following conditions hold:
  - The reset function is correct, i.e., it always brings back to the same state (fair)
  - The IUT is deterministic, fully specified, and observable (fair)
  - The IUT has the same number of states as the specification (hard)
- Complexity
  - Number of test cases proportional to $|P| \cdot |W|$, i.e., $O(N^2 \cdot N) = O(N^3)$
  - Would not be a problem, if there were not extra-states

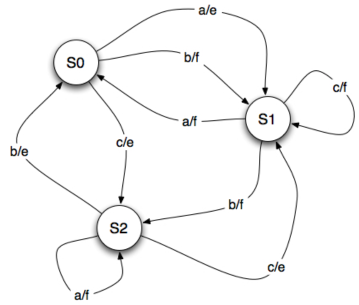


The WP-method: mitigate complexity (1/2)

- Simple concept
  - After the state cover $Q \times W$, transition cover $P \times W$ can be simplified into $(P \setminus Q) \times W$
- More clever concept
  - After any test prefix in Q or in $(P \setminus Q)$, the suffix of the full characterization set W can be replaced with an identification set restricted to the expected reached state
  - Replace the concept of a single characterization set W, with a collection of state charaterization sets, one for each state of the specification

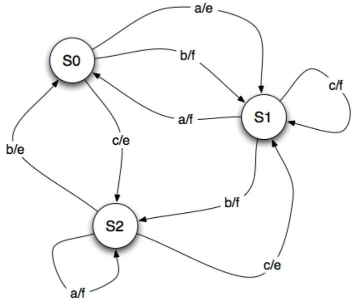
The WP-method: mitigate complexity (2/2)

- State characterization sets
  - E.g., $WP = \{W_0; W_1; W_2\} = \{\{a\}; \{a, b\}; \{b\}\}$
  - All test prefixes terminating in S_0 or S_1 will be extended by 1 single prefix rather than 2
  - Complexity is reduced from $O(N^3)$ to $O(N^2)$



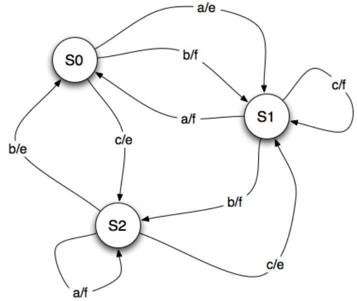
The WP-method: about assumptions (1/2)

- Specification
  - Deterministic: in the practice, ça va sans dire (but, in any case, this can be obtained by determinization)
  - Observable: can be obtained with a mute symbol
  - Completely specified: can be obtained with a mute self loop
  - Yet, each of these increases complexity, in the number of states, in the length or size of characterization sets



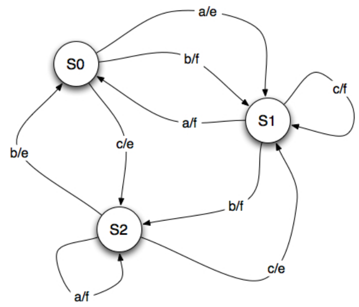
The WP-method: about assumptions (2/2)

- Implementation
  - Deterministic: is a constraint for coding
  - Observable: can be obtained with a mute symbol
  - Completely specified: the IUT must catch any input in any state, possibly with a No OPeration (NOP)
  - Correct reset function: requires that no memory is maintained, can be not trivial to implement, but can be tested
  - No extra states: the real limit!



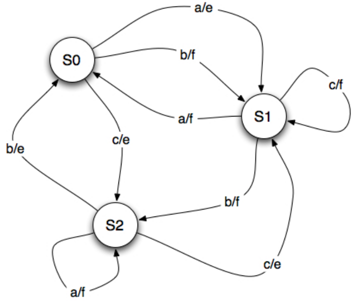
The WP-method: about extra states (1/2)

- What is the problem with extra states?
  - Equivalent extra states are not a fault
  - But, an extra state can replace a state, and the IUT can produce the same outputs as the specification by traversing extra states. . . for a number of steps longer than the test cases in the characterization set



The WP-method: about extra states (2/2)

- The solution
  - Absence of extra-states is not easy to guarantee in the implementation
  - Tests in the characterization set W can be extended to guarantee full coverage up to k extra states by replacing W with $I_k \times W$, where I_k is the set of all input sequences of length k
- Trade-off between complexity and fault coverage
  - Would require some estimation of the number of extra states
  - Full fault coverage is an ambitious aim


S0
S1
S2
a/e
b/f
c/f
a/f
c/e
b/e
b/f
c/e
a/f